

# **FROM DEVELOPMENT TO DEPLOYMENT: AGENTIC AUTOMATED AIOT APPLICATION LIFECYCLE**

SHEN LEMING

PhD

The Hong Kong Polytechnic University

2026

The Hong Kong Polytechnic University  
Department of Computing

# From Development to Deployment: Agentic Automated AIoT Application Lifecycle

SHEN Leming

A thesis submitted in partial fulfilment of the requirements  
for the degree of Doctor of Philosophy

May 2026

## CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

\_\_\_\_\_ (Signed)

SHEN Leming  
\_\_\_\_\_ (Name of student)



# Abstract

The Artificial Intelligence of Things (AIoT) has emerged as a transformative paradigm, integrating Artificial Intelligence (AI) techniques with the Internet of Things (IoT) to enable intelligent perception and decision-making in the physical world. However, the development and deployment of AIoT applications present significant barriers, demanding specialized expertise in signal processing, deep learning, and embedded systems to orchestrate heterogeneous hardware and diverse data modalities. While Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, their direct application to the specialized and resource-constrained domain of AIoT remains limited by a lack of domain knowledge, privacy concerns, and rigid development workflows.

This thesis presents a comprehensive framework for Agentic Automated AIoT Application Development, fully leveraging the reasoning and generative capabilities of LLMs to automate the entire lifecycle of AIoT applications from natural language programming to on-device deployment. First, we introduce **AutoIoT**, an LLM-driven system that democratizes AIoT programming by translating natural language requirements into executable code through automated knowledge retrieval and iterative refinement. Second, addressing the privacy and latency issues of cloud-based LLMs, we propose **GPIoT**, which tailors local Small Language Models (SLMs) for IoT program synthesis using a novel Parameter-Efficient Co-Tuning (PECT) paradigm to mitigate domain misalignment.

Furthermore, recognizing that fixed agentic workflows for AIoT application development struggle with evolving requirements and heterogeneous AIoT environment, we propose **SE<sup>2</sup>**, a self-exploring and self-evolving agentic paradigm. **SE<sup>2</sup>** aban-

dons rigid workflows in favor of a dynamic coordinator-actor architecture that autonomously constructs adaptive workflows based on real-time reasoning and mobile runtime feedback. Finally, moving from programming to deployment, we present **AutoDeploy**, an end-to-end system that resolves the critical “model-engine mismatch” issue and automates AIoT model deployment on heterogeneous embedded devices. By employing agents to analyze operator compatibility and utilizing a greedy cross-engine inference algorithm, **AutoDeploy** significantly reduces deployment time and inference latency.

Collectively, these contributions establish a robust foundation for the next generation of automated, accessible, and adaptive AIoT development.

## Publications Arising From the Thesis

1. **Leming Shen**, Qiang Yang, Yuanqing Zheng, and Mo Li. “AutoIOT: LLM-Driven Automated Natural Language Programming for AIoT Applications.” In Proceedings of the 31st Annual International Conference on Mobile Computing and Networking (MobiCom 2025), pp. 468–482, 2025.
2. **Leming Shen**, Qiang Yang, Xinyu Huang, Zijing Ma, and Yuanqing Zheng. “GPIoT: Tailoring Small Language Models for IoT Program Synthesis and Development.” In Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems (SenSys 2025), pp. 199–212, 2025.
3. **Leming Shen**, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. “FedConv: A Learning-on-Model Paradigm for Heterogeneous Federated Clients.” In Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services (MobiSys 2024), pp. 398–411, 2024.
4. **Leming Shen**, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. “Hierarchical and Heterogeneous Federated Learning via a Learning-on-Model Paradigm,” in IEEE Transactions on Mobile Computing (TMC), vol. 24, no. 11, pp. 12103-12120, Nov. 2025.
5. Xinyu Huang, Qiang Yang, **Leming Shen**, Zijing Ma, and Yuanqing Zheng. “Jailbreaking Embodied LLMs via Action-level Manipulation.” In Proceedings of the 24th ACM Conference on Embedded Networked Sensor Systems (SenSys 2026), pp. 1–14, 2026.
6. Xinyu Huang, **Leming Shen**, Zijing Ma, and Yuanqing Zheng. “Towards Privacy-Preserving and Personalized Smart Homes via Tailored Small Language Models.” IEEE Transactions on Mobile Computing (2026).
7. Kaiyan Cui, **Leming Shen**, Yuanqing Zheng, Fu Xiao, and Jinsong Han. “Talk2Radar: Talking to mmWave Radars via Smartphone Speaker.” In IEEE

INFOCOM 2024-IEEE Conference on Computer Communications, pp. 2358-2367. IEEE, 2024.

8. Kaiyan Cui, Qiang Yang, **Leming Shen**, Yuanqing Zheng, Fu Xiao, and Jinsong Han. “Towards ISAC-empowered mmWave Radars by Capturing Modulated Vibrations.” in IEEE Transactions on Mobile Computing (TMC), vol. 23, no. 12, pp. 13787-13803, Dec. 2024.

# Acknowledgements

Writing this thesis marks the conclusion of a long, challenging, yet incredibly rewarding journey. The path of research is rarely a straight line; it is paved with failed experiments, sleepless nights, and moments of doubt, but also with the unparalleled joy of discovery and the satisfaction of solving problems that once seemed insurmountable. This dissertation, while bearing my name, is not a solitary achievement. It is the result of years of cultivation, collaboration, and the unwavering support of many brilliant minds and kind hearts who have walked this path alongside me.

First and foremost, I would like to express my deepest and most sincere gratitude to my supervisor, Prof. Yuanqing Zheng. He has been much more than an academic advisor; he has been a mentor, a role model, and a constant source of inspiration throughout my PhD study. I am profoundly grateful for the trust he placed in me, providing not only the generous financial support and top-tier resources necessary for my experiments but, more importantly, the intellectual freedom to explore the emerging intersection of Large Language Models and AIoT. Prof. Zheng's vision and rigorous academic standards have fundamentally shaped my identity as a researcher. He taught me that research is not merely about publishing papers, but about cultivating a relentless pursuit of truth and striving for excellence in every detail. I will always cherish our discussions, whether they were deep dives into technical bottlenecks or broader conversations about career and life, where his sharp insights often turned my confusion into clarity. His patience during my periods of struggle and his encouragement when I lost confidence gave me the strength to persevere. I am truly fortunate to have started my academic career under his wing.

I am also deeply grateful to my visiting supervisor, Prof. Chris Xiaoxuan Lu,

for graciously hosting me at University College London during the final six months of my doctoral studies. His profound expertise and insightful guidance significantly broadened my research perspective, particularly during the development of our work on adaptive agentic workflows and robotic systems. I truly appreciate his mentorship and the stimulating academic environment he provided, which were instrumental in shaping my thesis.

I would like to extend my sincere thanks to the many external professors whose collaboration has enriched my work. I thank Prof. Mo Li (Hong Kong University of Science and Technology), Prof. Jinsong Han (Zhejiang University), Prof. Xiao-Yong Wei (Sichuan University and The Hong Kong Polytechnic University), Prof. Yaxiong Xie (University at Buffalo), and Prof. Fu Xiao (Nanjing University of Posts and Telecommunications). Their diverse perspectives and valuable suggestions helped me broaden my understanding of the field and refine the contributions of this thesis.

I am extensively grateful to Dr. Qiang Yang from the University of Cambridge. As a close collaborator on almost all my works (from `AutoIOT` to `AutoDeploy`), his sharp insights, constructive feedback, and willingness to engage in deep technical discussions have significantly elevated the quality of my research. I have learned a great deal from his dedication to academic excellence. I am also grateful to Prof. Kaiyan Cui from Nanjing University of Posts and Telecommunications. She has been much more than a brilliant collaborator during our time working together; her genuine friendship and the personal support she provided have made the rigorous demands of my PhD journey far more manageable.

My time in the research lab has been made memorable by the support and camaraderie of my colleagues. I am particularly thankful to Dr. Xianjin Xia, Dr. Shiming Yu, Dr. Jianwei Liu (Zhejiang University), Mr. Xinyu Huang, Mr. Zijing Ma, Mr. Xin Li, and Mr. Jianing Zhang for their direct contributions to our joint works, as well as for the countless hours we spent debugging code and discussing ideas. I also want to thank the other members of our research group, including Dr. Ningning Hou, Mr. Boan Yang, Miss Linya Fu, and Mr. Zehua Cao, for creating a stimulating and collaborative research environment where ideas could flourish.

Finally, I owe a debt of gratitude to my family and friends. To my parents, thank you for your unconditional love, sacrifices, and belief in my potential, even when the path was difficult. Your encouragement gave me the strength to persevere and reach this milestone.

As I close this chapter and look toward the future, I carry with me not just the technical knowledge documented in these pages, but the resilience and critical thinking instilled in me during this process. I hope that the work presented here contributes, even in a small way, to the advancement of our field, and I look forward to the new challenges and opportunities that lie ahead.



# Table of contents

|                                                                                           |            |
|-------------------------------------------------------------------------------------------|------------|
| <b>List of Figures</b>                                                                    | <b>xix</b> |
| <b>List of Tables</b>                                                                     | <b>xxv</b> |
| <b>1 Introduction</b>                                                                     | <b>1</b>   |
| 1.1 Background and Motivation . . . . .                                                   | 1          |
| 1.2 Challenges & Research Gaps . . . . .                                                  | 3          |
| 1.3 Thesis Contributions . . . . .                                                        | 4          |
| 1.4 Thesis Organization . . . . .                                                         | 6          |
| <b>2 Literature Review</b>                                                                | <b>9</b>   |
| 2.1 Overview . . . . .                                                                    | 9          |
| 2.2 The Evolution of Code Generation: From LLMs to Agentic AIoT<br>Development . . . . .  | 9          |
| 2.3 Localized Intelligence and Hardware-Aware Deployment . . . . .                        | 10         |
| 2.4 The Paradigm Shift: Adaptive and Self-Evolving Workflows in AIoT .                    | 11         |
| 2.5 The Last Mile: Deploying and Adapting AI Models on Heterogeneous<br>Devices . . . . . | 12         |
| <b>3 LLM-Driven Automated Natural Language Programming for AIoT<br/>    Applications</b>  | <b>13</b>  |
| 3.1 Introduction . . . . .                                                                | 13         |
| 3.2 Background & Motivation . . . . .                                                     | 17         |
| 3.2.1 Background . . . . .                                                                | 17         |

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| 3.2.2    | Preliminary Experiments . . . . .                             | 18        |
| 3.2.3    | Motivation & Key Ideas . . . . .                              | 21        |
| 3.3      | System Overview . . . . .                                     | 21        |
| 3.4      | System Design . . . . .                                       | 24        |
| 3.4.1    | User Interface . . . . .                                      | 24        |
| 3.4.2    | Background Knowledge Retrieval . . . . .                      | 25        |
| 3.4.3    | Automated Program Synthesis . . . . .                         | 27        |
| 3.4.4    | Code Improvement . . . . .                                    | 30        |
| 3.5      | Experiment Setup . . . . .                                    | 32        |
| 3.5.1    | Implementation . . . . .                                      | 32        |
| 3.5.2    | AIoT Applications & Datasets . . . . .                        | 32        |
| 3.6      | Evaluation . . . . .                                          | 34        |
| 3.6.1    | Metrics . . . . .                                             | 34        |
| 3.6.2    | Performance against Baselines . . . . .                       | 34        |
| 3.6.3    | Sensitivity Analysis . . . . .                                | 38        |
| 3.6.4    | Ablation Study . . . . .                                      | 41        |
| 3.6.5    | User Study . . . . .                                          | 43        |
| 3.7      | Discussion . . . . .                                          | 45        |
| 3.8      | Related Work . . . . .                                        | 47        |
| 3.9      | Chapter Summary . . . . .                                     | 49        |
| <b>4</b> | <b>Localizing Small Language Models for AIoT Applications</b> | <b>51</b> |
| 4.1      | Introduction . . . . .                                        | 51        |
| 4.2      | Background & Motivation . . . . .                             | 55        |
| 4.2.1    | Code LLM and LLM+RAG . . . . .                                | 56        |
| 4.2.2    | Preliminary Experiments & Findings . . . . .                  | 57        |

|          |                                                                             |           |
|----------|-----------------------------------------------------------------------------|-----------|
| 4.3      | System Overview . . . . .                                                   | 60        |
| 4.4      | System Design . . . . .                                                     | 61        |
| 4.4.1    | Data Collection & Augmentation . . . . .                                    | 61        |
| 4.4.2    | Parameter-Efficient Co-Tuning (PECT) . . . . .                              | 70        |
| 4.4.3    | Requirement Transformation . . . . .                                        | 74        |
| 4.5      | Experiment Setup . . . . .                                                  | 76        |
| 4.5.1    | Implementation . . . . .                                                    | 76        |
| 4.5.2    | IoT Applications . . . . .                                                  | 76        |
| 4.6      | Evaluation . . . . .                                                        | 77        |
| 4.6.1    | Metrics . . . . .                                                           | 77        |
| 4.6.2    | Baselines . . . . .                                                         | 78        |
| 4.6.3    | Application Evaluation . . . . .                                            | 78        |
| 4.6.4    | Breakdown Evaluation . . . . .                                              | 83        |
| 4.6.5    | Ablation Study . . . . .                                                    | 86        |
| 4.6.6    | User Study . . . . .                                                        | 89        |
| 4.7      | Discussion . . . . .                                                        | 90        |
| 4.8      | Related Work . . . . .                                                      | 91        |
| 4.9      | Chapter Summary . . . . .                                                   | 92        |
| <b>5</b> | <b>Fully Autonomous and Adaptive Agentic Workflows for AIoT Development</b> | <b>95</b> |
| 5.1      | Introduction . . . . .                                                      | 95        |
| 5.2      | Background & Motivation . . . . .                                           | 99        |
| 5.2.1    | Limitations of SOTA Methods . . . . .                                       | 99        |
| 5.2.2    | Motivation & Feasibility Study . . . . .                                    | 101       |
| 5.3      | SE <sup>2</sup> Overview . . . . .                                          | 104       |
| 5.4      | Detailed Design of SE <sup>2</sup> . . . . .                                | 105       |

|          |                                                                     |            |
|----------|---------------------------------------------------------------------|------------|
| 5.4.1    | Coordinator-Actor Agentic Architecture . . . . .                    | 105        |
| 5.4.2    | Mobile Runtime Feedback Mechanism . . . . .                         | 108        |
| 5.4.3    | Continuous LLM Tuning Paradigm . . . . .                            | 111        |
| 5.5      | Experiment Setup . . . . .                                          | 116        |
| 5.5.1    | Implementation . . . . .                                            | 116        |
| 5.5.2    | Evaluation Protocols . . . . .                                      | 117        |
| 5.6      | Evaluation . . . . .                                                | 119        |
| 5.6.1    | Benchmark Evaluation . . . . .                                      | 119        |
| 5.6.2    | Case Study on Edge AI Applications . . . . .                        | 121        |
| 5.6.3    | Ablation Study . . . . .                                            | 126        |
| 5.6.4    | User Study for Generalizability Test . . . . .                      | 127        |
| 5.7      | Discussion . . . . .                                                | 130        |
| 5.8      | Related Work . . . . .                                              | 130        |
| 5.9      | Chapter Summary . . . . .                                           | 131        |
| <b>6</b> | <b>Automating AIoT Model Deployment on Heterogeneous Mobile De-</b> | <b>133</b> |
|          | <b>vices</b>                                                        |            |
| 6.1      | Introduction . . . . .                                              | 133        |
| 6.2      | Background & Motivation . . . . .                                   | 138        |
| 6.2.1    | Embedded Model Deployment . . . . .                                 | 138        |
| 6.2.2    | Model-Engine Mismatch . . . . .                                     | 139        |
| 6.2.3    | AutoDeploy & Motivation . . . . .                                   | 141        |
| 6.3      | System Overview . . . . .                                           | 142        |
| 6.3.1    | AutoDeploy Workflow . . . . .                                       | 142        |
| 6.3.2    | Problem Formulation . . . . .                                       | 144        |
| 6.4      | System Design . . . . .                                             | 145        |
| 6.4.1    | Agent-Based Operator Analyzer . . . . .                             | 145        |

|          |                                                      |            |
|----------|------------------------------------------------------|------------|
| 6.4.2    | Structure-Aware Pattern Detector . . . . .           | 149        |
| 6.4.3    | Greedy Cross-Engine Inference Algorithm . . . . .    | 152        |
| 6.5      | Experiment Setup . . . . .                           | 157        |
| 6.5.1    | Implementation . . . . .                             | 157        |
| 6.5.2    | Evaluation Protocols . . . . .                       | 158        |
| 6.6      | Evaluation . . . . .                                 | 159        |
| 6.6.1    | Metrics . . . . .                                    | 159        |
| 6.6.2    | Baselines . . . . .                                  | 160        |
| 6.6.3    | Comprehensive Model Evaluation . . . . .             | 161        |
| 6.6.4    | Case Study: Embedded AI Applications . . . . .       | 163        |
| 6.6.5    | Ablation Study . . . . .                             | 167        |
| 6.6.6    | User Study for Generalizability Evaluation . . . . . | 168        |
| 6.7      | Discussion . . . . .                                 | 170        |
| 6.8      | Related Work . . . . .                               | 171        |
| 6.9      | Chapter Summary . . . . .                            | 172        |
| <b>7</b> | <b>Discussions</b>                                   | <b>173</b> |
| <b>8</b> | <b>Conclusions</b>                                   | <b>177</b> |
| <b>9</b> | <b>Appendix</b>                                      | <b>181</b> |
| 9.1      | Complexity Analysis . . . . .                        | 181        |
| 9.2      | Edge AI User Requirements . . . . .                  | 188        |
| 9.2.1    | Multimodal HAR User Requirement . . . . .            | 188        |
| 9.2.2    | Multimodal VLU User Requirement . . . . .            | 189        |
| 9.2.3    | Multimodal ODAD User Requirement . . . . .           | 190        |
| 9.3      | Agentic Workflow Comparison . . . . .                | 191        |
| 9.3.1    | Baseline Workflows . . . . .                         | 191        |

|       |                                             |            |
|-------|---------------------------------------------|------------|
| 9.3.2 | SE <sup>2</sup> Workflow for HAR . . . . .  | 193        |
| 9.3.3 | SE <sup>2</sup> Workflow for VLU . . . . .  | 195        |
| 9.3.4 | SE <sup>2</sup> Workflow for ODAD . . . . . | 197        |
| 9.4   | Generated Code Comparison . . . . .         | 198        |
| 9.4.1 | Sample Code on HAR . . . . .                | 198        |
| 9.4.2 | Sample Code on VLU . . . . .                | 200        |
| 9.4.3 | Sample Code on ODAD . . . . .               | 202        |
| 9.5   | Agentic Recovery Mechanism . . . . .        | 204        |
| 9.6   | More Ablation Studies . . . . .             | 205        |
|       | <b>References</b>                           | <b>207</b> |

# List of Figures

|      |                                                                                                                                                                                                                                                                                                                                                                                                                                                            |    |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1  | Illustration of how LLMs can sense and interact with the physical world in AIoT applications. . . . .                                                                                                                                                                                                                                                                                                                                                      | 14 |
| 3.2  | Prior work processes sensor data with LLMs. . . . .                                                                                                                                                                                                                                                                                                                                                                                                        | 17 |
| 3.3  | An example of direct code generation. . . . .                                                                                                                                                                                                                                                                                                                                                                                                              | 18 |
| 3.4  | Code generation with user intervention. . . . .                                                                                                                                                                                                                                                                                                                                                                                                            | 20 |
| 3.5  | The system overview and workflow of <b>AutoIoT</b> . . . . .                                                                                                                                                                                                                                                                                                                                                                                               | 22 |
| 3.6  | <b>AutoIoT</b> agent answers user’s prompt with LLMs and LangChain tools. . . . .                                                                                                                                                                                                                                                                                                                                                                          | 23 |
| 3.7  | The prompt template for (a) terminology determination, (b) terminology searching, (c) algorithm outline generation, and (d) detailed design generation. . . . .                                                                                                                                                                                                                                                                                            | 25 |
| 3.8  | The prompt template for (a) modularized code generation, (b) modularized code integration, (c) code debugging, and (d) iterative code improvement. . . . .                                                                                                                                                                                                                                                                                                 | 29 |
| 3.9  | An example of code improvement via iterations (details omitted). . . . .                                                                                                                                                                                                                                                                                                                                                                                   | 31 |
| 3.10 | The overall performance of the four IoT applications. In (a), A. for <b>AutoIoT</b> , H. for Hamilton, C. for Christov, E. for Engzee, P. for Pantompkins, and S. for SWT. In (b), N. for NN, 1D for 1D-CNN, Bi. for BiLSTM, C. for Conv-LSTM, L. for LSTM-RNN, and <b>n</b> for ResNet- <b>n</b> . In (c) & (d), A.1, A.2, and A.3 for three different <b>AutoIoT</b> -generated programs; B. for the baseline in the multimodal HAR application. . . . . | 35 |
| 3.11 | Different levels of user intervention. . . . .                                                                                                                                                                                                                                                                                                                                                                                                             | 38 |
| 3.12 | Different numbers of iteration. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                    | 39 |
| 3.13 | Different LLMs. (4 for GPT-4, 3.5 for GPT-3.5, C. for Cohere, A. for Anthropic Claude 2, L. for Llama2-7b, and G. for Gemma-7b.) . . . .                                                                                                                                                                                                                                                                                                                   | 40 |

|      |                                                                                                                                                                                   |    |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.14 | Ablation study. (B – background knowledge retrieval, O. – algorithm outline generation, D. – detailed design generation, I. – code improvement.) . . . . .                        | 41 |
| 3.15 | User study (a & b: objective evaluation; c: subjective evaluation) . .                                                                                                            | 43 |
| 3.16 | Further experiments. . . . .                                                                                                                                                      | 45 |
| 4.1  | Existing code LLMs need to transmit sensitive data to remote servers. In contrast, <b>GPIoT</b> features three local SLMs to protect user privacy and reduce query costs. . . . . | 52 |
| 4.2  | (a) Existing LLMs tend to prioritize general terms; (b) LLM+RAG systems require multiple cascaded agents. . . . .                                                                 | 56 |
| 4.3  | (a) Directly tuning SLM with simple augmentation only yields small improvements; (b) Gap between SLMs. . . . .                                                                    | 58 |
| 4.4  | The system overview and workflow of <b>GPIoT</b> (All the local SLMs share the same foundation model). . . . .                                                                    | 60 |
| 4.5  | Task decomposition dataset construction. . . . .                                                                                                                                  | 63 |
| 4.6  | Prompts for paper information extraction. . . . .                                                                                                                                 | 63 |
| 4.7  | Examples of IoT-oriented data augmentation with different modalities, representations, and resource budgets . . . . .                                                             | 64 |
| 4.8  | (a) Tuning data sample for task decomposition. (b) Prompt for sensor modality augmentation. . . . .                                                                               | 65 |
| 4.9  | An example of a Python package’s website. . . . .                                                                                                                                 | 66 |
| 4.10 | (a) & (b) Two tuning data samples. (c) Well-structured task specification. d) Traditional LoRA tuning paradigm. . . . .                                                           | 68 |
| 4.11 | PECT in one Transformer block with both independently and collaboratively tuned LoRA adapters. . . . .                                                                            | 71 |
| 4.12 | CoT-based prompts for RTSLM. . . . .                                                                                                                                              | 75 |
| 4.13 | Problem statements of the two applications (HAR and multimodal HAR share a similar prompt). . . . .                                                                               | 79 |
| 4.14 | The overall performance of HD. . . . .                                                                                                                                            | 79 |
| 4.15 | The overall performance of HAR. . . . .                                                                                                                                           | 80 |

|      |                                                                                                                                                                                                                              |     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 4.16 | The overall performance of multimodal HAR . . . . .                                                                                                                                                                          | 82  |
| 4.17 | Breakdown evaluation on TDSLM. . . . .                                                                                                                                                                                       | 84  |
| 4.18 | Breakdown evaluation on CGSLM . . . . .                                                                                                                                                                                      | 85  |
| 4.19 | (a) Evaluating <b>GPIoT</b> 's performance using prompts with different resource constraints. (b) Evaluating the quality of the code generated by CGSLM using SonarQube. . . . .                                             | 85  |
| 4.20 | Ablation of IoT-oriented augmentation (A.). . . . .                                                                                                                                                                          | 86  |
| 4.21 | Ablation of PECT (P. in the figure). . . . .                                                                                                                                                                                 | 87  |
| 4.22 | Ablation of requirement transformation. . . . .                                                                                                                                                                              | 88  |
| 4.23 | User study on different tasks. . . . .                                                                                                                                                                                       | 90  |
| 5.1  | Previous works handcrafted fixed agentic workflows to synthesize programs. <b>SE</b> <sup>2</sup> autonomously explores and selects the next atomic operation in program synthesis to meet diverse edge AI requirements. . . | 96  |
| 5.2  | A typical agentic workflow of existing work (Step 3 of the workflow is detailed in the figure). . . . .                                                                                                                      | 100 |
| 5.3  | Preliminary & feasibility study. . . . .                                                                                                                                                                                     | 101 |
| 5.4  | The overall architecture and workflow of <b>SE</b> <sup>2</sup> (with detailed operations in step 3). . . . .                                                                                                                | 104 |
| 5.5  | Ensemble mutual-actor-critic strategy. . . . .                                                                                                                                                                               | 113 |
| 5.6  | Real-world experiment setup. . . . .                                                                                                                                                                                         | 117 |
| 5.7  | Performance comparison on <i>IoT Bench</i> (SE: <b>SE</b> <sup>2</sup> , GC: GitHub Copilot, MC: MapCoder, AC: AgentCoder, AS: AgentSquare, AI: AutoIOT, GT: GPIoT, IP: IoTPilot, EG: EmbedGenius). . . . .                  | 120 |
| 5.8  | Performance comparison on HAR (SE – <b>SE</b> <sup>2</sup> , GC – GitHub Copilot, MC – MapCoder, AC – AgentCoder, AS – AgentSquare, AI – AutoIOT, GT – GPIoT, IP – IoTPilot, EG – EmbedGenius). . . . .                      | 122 |
| 5.9  | Performance comparison on VLU. . . . .                                                                                                                                                                                       | 123 |
| 5.10 | Performance comparison on ODAD. . . . .                                                                                                                                                                                      | 125 |

|      |                                                                                                                                                                                                                                                      |     |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| 5.11 | Ablation study (F. – feedback mechanism, E. – ensemble mutual-actor-critic strategy, S. – self-reflective grader, C. – continuous LLM tuning). . . . .                                                                                               | 126 |
| 5.12 | Evaluation results of the user study. . . . .                                                                                                                                                                                                        | 128 |
| 6.1  | Left: a typical workflow of embedded AI model deployment and the <i>model-engine mismatch</i> issue. Right: the high-level idea of <b>AutoDeploy</b> . . . . .                                                                                       | 135 |
| 6.2  | (a) The model structure. (b) The model deployment success rate across various types of models and inference engines (TensorRT does not support CPU mode). (ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO, MI – MILR) . . . . .                   | 138 |
| 6.3  | The mismatch between AI models and engines. . . . .                                                                                                                                                                                                  | 139 |
| 6.4  | The system overview and workflow of <b>AutoDeploy</b> with three novel technical modules. . . . .                                                                                                                                                    | 142 |
| 6.5  | (a) The vanilla partitioning method results in significantly longer inference time (ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO); (b) An example of operator optimization: merging the <b>Conv</b> and the <b>Add</b> operator. . . . .        | 150 |
| 6.6  | (a) Inference latency when randomly assigning compatible inference engines; (b) The three engines require distinct times to execute the three common operators (Gemm – general matrix multiplication). . . . .                                       | 152 |
| 6.7  | The <i>greedy cross-engine inference algorithm</i> . . . . .                                                                                                                                                                                         | 154 |
| 6.8  | Evaluation results on the 150 open-source models (Left: GPU runtime; Right: CPU runtime; Labels from bottom to top: AD – <b>AutoDeploy</b> , ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO, MI – MLIR, IP – IoTPilot, EG – EmbedGenius). . . . . | 161 |
| 6.9  | Evaluation results for image recognition (AD – <b>AutoDeploy</b> , H1 – human group 1, H2 – human group 2, H3 – human group 3, IP – IoTPilot, EG – EmbedGenius, ORT – ONNX Runtime, OV – OpenVINO, MI – MILR). . . . .                               | 163 |
| 6.10 | Evaluation results for speech sentiment analysis (SSA). . . . .                                                                                                                                                                                      | 165 |
| 6.11 | Evaluation results for vision-language understanding (VLU). . . . .                                                                                                                                                                                  | 166 |
| 6.12 | Ablation study on speech sentiment analysis (AOA – Agent-Based Operator Analyzer, SPD – Structure-Aware Pattern Detector, PC – pattern credit, SP – switching penalty). . . . .                                                                      | 167 |

|      |                                                                    |     |
|------|--------------------------------------------------------------------|-----|
| 6.13 | Evaluation results of the user study. . . . .                      | 169 |
| 9.1  | The agentic workflow of MapCoder [1]. . . . .                      | 191 |
| 9.2  | The agentic workflow of AgentCoder [2]. . . . .                    | 191 |
| 9.3  | The agentic workflow of AgentSquare [3]. . . . .                   | 191 |
| 9.4  | The agentic workflow of AutoIOT [4]. . . . .                       | 192 |
| 9.5  | The agentic workflow of GPIoT [5]. . . . .                         | 192 |
| 9.6  | The agentic workflow of IoTPilot [6]. . . . .                      | 192 |
| 9.7  | The agentic workflow of EmbedGenius [7]. . . . .                   | 192 |
| 9.8  | Sample code comparison on the multimodal HAR application. . . . .  | 199 |
| 9.9  | Sample code comparison on the multimodal VLU application. . . . .  | 201 |
| 9.10 | Sample code comparison on the multimodal ODAD application. . . . . | 203 |
| 9.11 | Ablation study on VLU. . . . .                                     | 205 |
| 9.12 | Ablation study on ODAD. . . . .                                    | 206 |



# List of Tables

|     |                                                    |     |
|-----|----------------------------------------------------|-----|
| 3.1 | Participant information of user study (N=20)       | 43  |
| 3.2 | AutoIoT vs SOTA code LLMs                          | 48  |
| 5.1 | Participant information of user study (N=20)       | 127 |
| 6.1 | Detailed information of the 150 downloaded models. | 140 |
| 6.2 | Detailed information of the embedded devices.      | 157 |
| 6.3 | Participant information of the user study (N=20)   | 168 |



# Chapter 1

## Introduction

### 1.1 Background and Motivation

The convergence of Artificial Intelligence (AI) and the Internet of Things (IoT) has birthed a transformative paradigm known as the Artificial Intelligence of Things (AIoT) [8, 9, 10, 11, 12, 13, 14, 15, 16]. Unlike traditional IoT, which primarily focuses on data collection and transmission, AIoT empowers edge devices with the intelligence to perceive, reason, and act upon the physical world in real-time. This synergy is driving a revolution across industries: from smart wearables that monitor health vitals to autonomous drones navigating complex environments and industrial robots optimizing manufacturing lines. The proliferation of these devices suggests a future where intelligence is ubiquitous, embedded in the very fabric of our daily lives [17, 18, 19, 20, 21].

Despite this immense potential, the development of robust AIoT applications remains the province of a select few experts. The creation of an AIoT system is inherently a “full-stack” challenge, demanding mastery over a fragmented and complex landscape. A developer must navigate through signal-model-deployment iteration:

- *Heterogeneous Hardware.* The ecosystem spans a vast array of devices, from resource-constrained microcontrollers (*e.g.*, Arduino [22], ESP32 [23]) to powerful edge accelerators (*e.g.*, NVIDIA Jetson [24], Raspberry Pis [25]), each

with unique instruction sets, memory limits, and peripheral interfaces [26, 27].

- *Multimodal Data Processing.* Real-world applications require integrating diverse signal modalities, including audio [28, 29], inertial measurement units (IMU) [30, 31], and vision [32, 33], each requiring specialized signal processing pipelines before deep learning models can even be applied.
- *Deployment Complexities.* Bridging the gap between a model trained in a high-performance cloud environment and a restricted edge runtime is notoriously difficult [34, 35]. Issues such as operator incompatibility, memory leaks, and latency constraints often force developers to rewrite code or manually optimize models for specific inference engines.

Consequently, the high barrier to entry suppresses innovation. Domain experts, such as doctors wanting to deploy gait analysis systems or civil engineers monitoring structural health, are often blocked by the sheer technical complexity of implementing the underlying AIoT software stack.

The recent emergence of Large Language Models (LLMs) has disrupted the field of software engineering. Their ability to generate syntactically correct code from natural language prompts offers a glimpse of a future where programming is democratized [36, 37]. However, simply applying off-the-shelf LLMs (like GPT-4 [38]) to the AIoT domain is insufficient. General-purpose models lack the “grounded” knowledge of physical sensors and specific hardware APIs. They often hallucinate libraries that do not exist for embedded platforms or generate Python code that cannot run on C++-based microcontrollers [4, 7]. Furthermore, relying on cloud-based LLMs for IoT development introduces critical privacy risks regarding sensitive sensor data and suffers from latency penalties that are unacceptable for real-time edge applications [5].

## 1.2 Challenges & Research Gaps

This thesis identifies and addresses four critical gaps that currently hinder the automated development of AIoT applications:

- *The Semantic Gap in Program Synthesis.* While LLMs excel at generic Python coding, they struggle with the specific semantics of IoT hardware [4, 39]. Transforming a high-level intent (*e.g.*, “Detect a fall using the accelerometer”) into executable code requires a complex chain of reasoning: selecting the right sampling rate, applying specific filters (*e.g.*, Butterworth filter [40]), and interfacing with the correct hardware driver. Existing “Copilot” tools [41] lack the domain-specific context to bridge this gap, often producing code that is logically sound but functionally useless on the target device [42].
- *Privacy and Efficiency in Cloud-Based Automation.* The dominant paradigm of sending user requirements and code context to cloud-based LLMs poses severe privacy risks, particularly when dealing with proprietary industrial designs or sensitive personal data [43]. Moreover, the high inference cost and latency of massive models make them impractical for continuous, iterative development loops [44]. There is a pressing need for local, privacy-preserving alternatives, yet small language models (SLMs) typically lack the reasoning capacity to handle complex AIoT logic.
- *Rigidity of Static Agentic Workflows.* Current automated software engineering agents [1, 2] including `AutoIOT` and `GPIOt` typically operate on fixed, static workflows (*e.g.*, “Retrieve”  $\rightarrow$  “Plan”  $\rightarrow$  “Code”  $\rightarrow$  “Test”). While effective for static tasks, these rigid pipelines are brittle in the dynamic world of AIoT. If a sensor is unavailable, or if the data modality shifts (*e.g.*, from camera to LiDAR), a static agent fails. Ideal automation requires agentic systems capa-

ble of both self-exploration and self-evolution, *i.e.*, agents that can dynamically alter their own workflows and continuously adapt to new heterogeneous environments, deciding when to backtrack, when to search for new information, and when to request human feedback.

- *The Model-Engine Mismatch in Deployment.* Even after an AI model is successfully developed and well-trained, deploying it to the edge is a major hurdle. A significant “model-engine mismatch” exists where operators used in modern neural networks are not supported by the diverse inference engines (*e.g.*, TensorRT [45], OpenVINO [46], TFLite [47]) running on edge devices. Developers are currently forced into a manual, trial-and-error process of operator registration or graph rewriting. There is no automated framework that can intelligently map model operators to the most efficient available execution backend without human intervention.

### 1.3 Thesis Contributions

To overcome these barriers, this thesis proposes a unified, intelligent framework for Agentic Automated AIoT Application Development. We leverage the reasoning capabilities of LLMs not just as coding tools, but as autonomous agents capable of planning, optimizing, and evolving AIoT systems. The specific contributions are:

- We introduce **AutoIOT**, an end-to-end framework that enables users to create AIoT applications using purely natural language. We propose a structure-aware retrieval mechanism that augments the LLM with up-to-date hardware documentation. Uniquely, **AutoIOT** implements a compiler-feedback loop. Instead of one-shot code generation, the system interprets error/execution logs from the edge device compilation process and autonomously iteratively debugs

the code, significantly boosting executability rates on mobile devices like Raspberry Pis [25].

- To solve the privacy and resource constraints, we present **GPIoT**, a system that tailors Small Language Models (SLMs) specifically for the IoT domain. We identify that SLMs struggle with “domain misalignment” when trying to be both general planners and specific coders. We introduce a *Parameter-Efficient Co-Tuning* (PECT) paradigm. This allows collaborative SLMs to share domain knowledge without bloating memory usage during tuning. We also construct the first large-scale instruction-tuning dataset dedicated to IoT tasks, enabling a 13-billion parameter model to rival the coding performance of far larger cloud models within this niche.
- Following **AutoIOT** and **GPIoT**, we address the rigidity of fixed agentic workflows with **SE<sup>2</sup>** (Self-Exploration and Self-Evolution). We move beyond fixed agentic workflows to a dynamic Coordinator-Actor architecture. **SE<sup>2</sup>** treats development steps as atomic actions (*i.e.*, **retrieve**, **reason**, **code**, and **test**). A Coordinator agent dynamically plans the workflow based on real-time feedback from the mobile runtime. If the agent encounters a new environment, it “evolves” by dynamically adding necessary atomic actions, allowing it to adapt to unseen sensor configurations and task requirements without human reprogramming.
- Finally, moving from code generation to model execution and deployment, we propose **AutoDeploy** to solve the *model-engine mismatch* issue. We treat deployment as a multi-agent negotiation problem. Agents analyze the source code of operators to determine compatibility with various inference engines. The system features a *Greedy Cross-Engine Inference* algorithm. It automat-

ically partitions a neural network, assigning different sub-graphs to the most efficient engine (*e.g.*, running convolutions on a GPU-optimized engine while offloading unsupported operators to a CPU fallback), reducing inference latency by over 26% compared to standard baselines.

Together, these four contributions interlock to form a cohesive, end-to-end pipeline that automates the entire **AutoIoT** application lifecycle. The lifecycle initiates with **AutoIoT**, which democratizes the development phase by seamlessly translating high-level natural language requirements into executable AIoT code through automated knowledge retrieval and iterative refinement. To ensure this initial programming process remains secure and resource-efficient, **GPIoT** localizes the code synthesis using tailored SLMs, directly mitigating the privacy risks and latency bottlenecks associated with cloud-based development. Because physical AIoT environments and hardware availability frequently change, **SE<sup>2</sup>** actively orchestrates this development phase by replacing rigid coding pipelines with a self-exploring, dynamic architecture that continuously adapts the workflow based on real-time mobile runtime feedback. Finally, **AutoDeploy** bridges the gap to the physical world, completing the lifecycle by taking the successfully synthesized AI models and automatically resolving critical model-engine mismatches to ensure optimized, low-latency deployment directly onto heterogeneous edge devices. Collectively, these systems forge a unified, agentic framework that autonomously navigates the complex transition from initial natural language conceptualization to robust, on-device execution.

## 1.4 Thesis Organization

The remainder of this thesis is structured as follows: Chapter 2 provides a comprehensive review of the state-of-the-art in LLM-driven program synthesis, autonomous agentic workflows, and efficient edge AI deployment. Chapter 3 presents the **AutoIoT**

framework, detailing the methodology for translating natural language into executable AIoT code and the feedback-driven debugging mechanism. Chapter 4 introduces **GPIoT**, focusing on the construction of IoT-specific datasets and the PECT tuning paradigm for efficient, private local synthesis. Chapter 5 elaborates on **SE<sup>2</sup>**, demonstrating how adaptive workflows and reinforcement strategies enable agents to handle dynamic AIoT environments. Chapter 6 describes **AutoDeploy**, explaining the operator analysis agents and the cross-engine optimization algorithms for heterogeneous devices. Chapter 7 concludes the thesis, summarizing the impact of these works and outlining future research directions for agentic AI systems. Chapter 8 includes some appendices as auxiliary contents and materials for the thesis.



# Chapter 2

## Literature Review

### 2.1 Overview

This chapter provides a comprehensive review of the literature bridging LLMs and AIoT. It traces the trajectory of AI-driven application development from foundational code generation models to sophisticated multi-agent software engineering frameworks. Subsequently, the chapter explores the critical transition of these coding agents into the physical realm of AIoT, detailing the shift toward localized Small Language Models (SLMs) for hardware-aware deployment. Finally, it surveys the current frontier of the field: the transition from rigid, rule-based execution pipelines to fully adaptive, autonomous agentic workflows capable of self-evolution in dynamic Edge AI environments.

### 2.2 The Evolution of Code Generation: From LLMs to Agentic AIoT Development

The advent of LLMs has fundamentally disrupted traditional software engineering. Foundational models such as Codex [48], CodeLlama [49], and StarCoder [50] demonstrated unprecedented capabilities in translating natural language intent into functional, executable code. However, efficiently transforming complex requirements into reliable software in a single generation step remains highly error-prone due to coding

hallucinations and logical omissions.

To mitigate these issues, the literature has seen a massive shift toward orchestrating LLMs as autonomous, collaborative agents. Frameworks such as ChatDev [51] apply LLMs to the traditional software waterfall model by simulating a virtual software company. In this multi-agent system (MAS), specialized agents (*e.g.*, CTO, programmer, tester) collaborate through “chat chains” and communicative de-hallucination to design, code, and test applications. Similarly, MetaGPT [52] encodes Standard Operating Procedures (SOPs) into agent prompts to streamline complex coding tasks, while SWE-agent [53] and GuggingGPT [54] provide LLMs with computer-use tools to autonomously navigate repositories and resolve GitHub issues. AutoGen [55] further generalized this paradigm by enabling customizable conversational patterns among agents for diverse coding workflows.

Crucially, substantial challenges arise when transitioning this agentic software engineering paradigm into the physical domain, particularly for AIoT applications involving heterogeneous embedded devices. Traditional embedded programming requires specialized knowledge of microcontrollers, sensor APIs, and resource constraints, which general-purpose coding agents often lack. Therefore, we propose AutoIOT [4], the first attempt to bridge this gap by establishing natural language programming interfaces for AIoT application development [6, 7].

## 2.3 Localized Intelligence and Hardware-Aware Deployment

While cloud-based LLM agents excel at synthesizing high-level application logic, their reliance on remote servers introduces severe vulnerabilities in IoT environments, including prohibitive query costs, unstable network dependencies, and strict data privacy violations [56, 57]. Consequently, the literature emphasizes moving

intelligence directly to the edge using Small Language Models (SLMs). SLMs (typically under 713 billion parameters) provide a resource-efficient alternative that facilitates localized code generation and workflow orchestration [58]. However, out-of-the-box SLMs often lack the specialized domain knowledge required for nuanced IoT coding. To address this, researchers leverage Parameter-Efficient Fine-Tuning (PEFT) methods [59], particularly Low-Rank Adaptation (LoRA) [60]. Therefore, we propose **GPIoT** [5] that highlights the necessity of fine-tuning these models on IoT-specialized datasets. Instead of using PEFT, we design a new tuning method, namely Parameter-Efficient Co-Tuning, which aligns multiple SLMs across the various stages of development (*i.e.*, task decomposition, code generation) to prevent domain misalignment and ensure high-fidelity, privacy-preserving local synthesis.

## 2.4 The Paradigm Shift: Adaptive and Self-Evolving Workflows in AIoT

The culmination of agent-driven AIoT development is the transition from fixed to fully adaptive workflows. In earlier systems like ChatDev or MetaGPT, agents operate on rigid, predefined sequences. While suitable for common development pipelines, they may not effectively handle highly volatile physical Edge AI environments characterized by missing sensor modalities, unpredictable hardware faults, and dynamic memory constraints. Fixed pipelines inevitably fail when faced with these runtime anomalies [3, 61, 62]. Therefore, we propose **SE<sup>2</sup>** that distills AIoT application development into four atomic operations (*i.e.*, **retrieve**, **reason**, **code**, **test**) and fully exploits the reasoning capabilities of LLMs to decide an appropriate next operation to dynamically and adaptively construct agentic workflows. This shift from static code generation to self-exploring, adaptive workflows represents the definitive foundation for autonomous AIoT application development.

## 2.5 The Last Mile: Deploying and Adapting AI Models on Heterogeneous Devices

Beyond synthesizing executable, high-performing AIoT programs, the literature identifies the physical deployment of compiled models onto heterogeneous embedded hardware as a major bottleneck. Pretrained models contain custom operators that frequently lack support across diverse on-device inference engines (*e.g.*, Intel’s OpenVINO [46], NVIDIA’s TensorRT [45]). This “model-engine mismatch” traditionally necessitates labor-intensive manual patches. Automating this deployment phase is critical. Therefore, we propose **AutoDeploy** that employs LLM agents to autonomously parse inference engine source code, evaluate operator compatibility, and dynamically partition models. By orchestrating a greedy cross-engine execution, such systems preserve optimizable execution patterns while completely removing the human developer from the hardware troubleshooting loop, seamlessly connecting localized code synthesis with efficient physical execution.

## Chapter 3

# LLM-Driven Automated Natural Language Programming for AIoT Applications

### 3.1 Introduction

Artificial Intelligence of Things (AIoT) [63, 64, 65, 66, 27, 67] is an emerging paradigm that leverages advanced artificial intelligence (AI) algorithms to process a vast amount of data generated by Internet of Things (IoT) devices. This technology brings a new level of intelligence and automation to various applications, including healthcare [68, 69], smart sensing [12, 28, 29], and autonomous driving [70].

Recent advances in large language models (LLMs) (*e.g.*, GPT-4 [38]) fundamentally changed the way we interact with AI. While initially designed to understand natural languages, recent pioneering works [71, 72, 73] have demonstrated considerable proficiency of LLMs in exploiting embedded world knowledge by interpreting IoT sensor data to perform various AIoT tasks. Recent works [71] term such an endeavor – Penetrative AI. Fig. 3.1 illustrates how LLMs can be tasked to comprehend and even interact with the physical world through integration with IoT sensors and actuators.

However, current LLMs on AIoT tasks [71, 72, 74] fall short in supporting AIoT

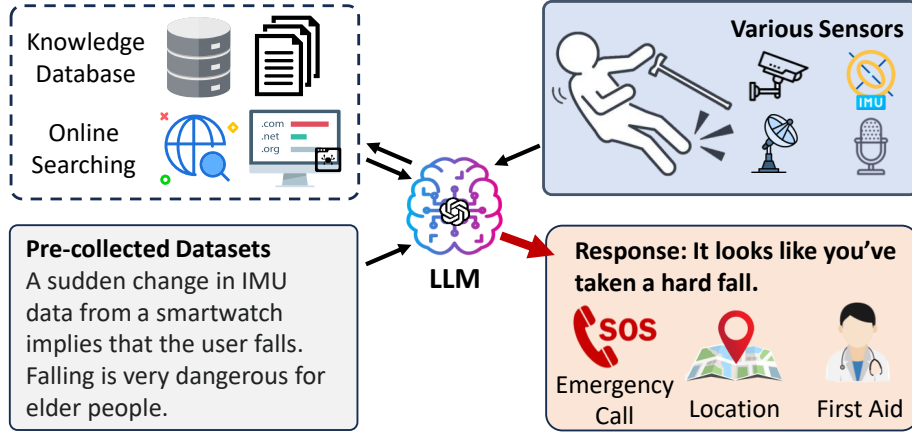


Figure 3.1: Illustration of how LLMs can sense and interact with the physical world in AIoT applications.

applications [75, 76, 77, 78, 79, 80, 81]: 1) The trustworthiness of the inference results is compromised since the inference process is performed inside a “black box” and opaque to users. Thus, the robustness of the applications or the correctness of the inference results are hard to verify; 2) Transmitting the raw or intermediate sensor data from user devices to LLM servers raises privacy concerns, incurs prohibitive query fees, and increases response latency; 3) Sensor data typically exhibits extensive length and high dimensionality, making remote processing at LLM servers infeasible due to token limits [82]. Ideally, the integration of LLMs with AIoT applications should be trustworthy, privacy-preserving, and communication-efficient.

On the other hand, existing works on LLMs have showcased their remarkable capabilities in code generation to accomplish a variety of programming tasks [83, 84, 85]. *Can we leverage LLMs to synthesize programs to fulfill AIoT application requirements?* This approach can 1) enhance the explainability and trustworthiness of the AIoT applications as the synthesized programs can be examined and interpreted by developers, 2) mitigate privacy concerns, and reduce the communication cost since the programs can be executed locally on user devices without offloading raw sensor data, and 3) efficiently process high-dimensional continuous sensor data without be-

ing limited by the token size or bounded by the round trip time over the network. To this end, we propose **AutoIOT**, a user-friendly natural language programming system based on LLMs. **AutoIOT** automatically identifies and retrieves the necessary domain knowledge over the Internet, intelligently synthesizes programs, and evolves the programs iteratively given sample inputs and ground truth. Surprisingly, we found that the synthesized programs can sometimes outperform some representative baselines and sample programs of recent academic papers.

While the automatic program synthesis for AIoT applications is promising and exciting, it entails tremendous technical challenges. 1) **High Complexity of AIoT Tasks**. Contrary to existing works that generate code for individual modules or well-defined functions [86], AIoT applications typically need a systematic design and integration involving multiple functional components, leading to much higher reasoning and planning complexity beyond the capability of current LLMs. To address this issue, **AutoIOT** decomposes the programming task into several distinct modules and generates the corresponding code segments. In particular, **AutoIOT** leverages chain-of-thought (CoT) prompts [87] to divide the task into a few sub-tasks and integrate their solutions, eventually making the sub-tasks manageable by LLMs. 2) **Lack of Domain Knowledge in AIoT**. LLMs are trained on pre-collected general corpus datasets, which may not include the latest domain-specific knowledge needed for the development of various emerging AIoT applications. To tackle this problem, **AutoIOT** guides the LLMs to search and retrieve necessary knowledge and algorithms, thereby enabling in-context training and inference augmented with domain knowledge for LLMs. 3) **Heavy Intervention and Constant Feedback**. Our preliminary experiments (§ 3.2.2) reveal that to generate functionally correct programs, developers have to give timely feedback to LLMs and constantly intervene in the entire development process. For example, developers need to provide specific reference materials and describe algorithms in great detail, which can be time-consuming and defeat

the very purpose of automated natural language programming. Ideally, **AutoIOT** should be able to synthesize the program with no intervention from users and require minimum user input only when necessary. To this end, we develop **AutoIOT** that can execute, debug, and optimize the synthesized program given sample inputs and outputs.

We fully implement **AutoIOT\*** and evaluate its synthesized programs with four representative AIoT applications: heartbeat detection, IMU-based human activity recognition (HAR), mmWave-based HAR, and multimodal HAR. Extensive experiments and user studies show that, the synthesized programs can achieve comparable performance to the corresponding baselines and significant improvements in user satisfaction. Besides, **AutoIOT** substantially reduces the communication cost and the total execution time. These findings demonstrate the LLM’s exceptional proficiency and great potential in synthesizing programs for AIoT applications. In summary, we make the following contributions:

- To our best, **AutoIOT** is the first work that enables systematic natural language programming for AIoT applications.
- We design and implement three novel technical modules (*i.e.*, background knowledge retrieval module, CoT prompting-based program synthesis module, and automated code improvement module) to synthesize and optimize programs for AIoT applications.
- Our comprehensive experiments demonstrate that synthesized programs can achieve comparable performance to baselines and sometimes outperform them.

---

\*The project is available at <https://github.com/lemingshen/AutoIOT>

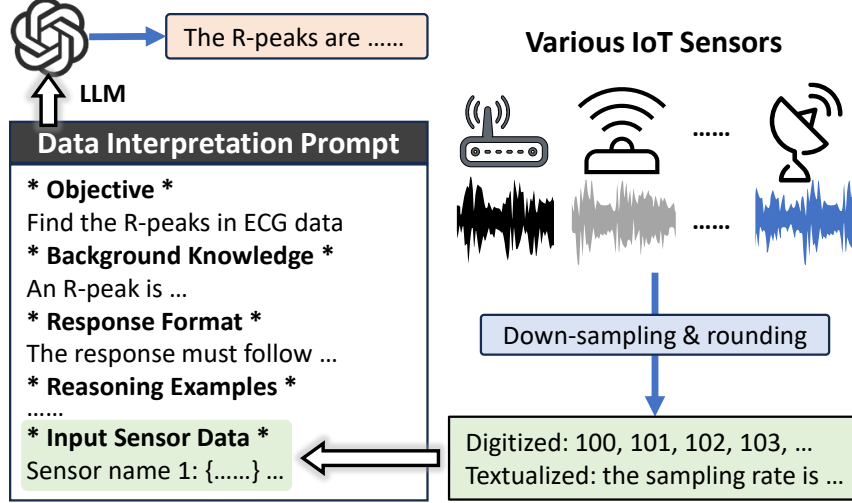


Figure 3.2: Prior work processes sensor data with LLMs.

## 3.2 Background & Motivation

We first revisit the pioneering Penetrative AI efforts that leverage the embedded knowledge of LLMs to address AIoT tasks. Then, we present the results of our preliminary experiments to demonstrate the feasibility and identify the key challenges of implementing an LLM-driven automated natural language programming system for AIoT applications.

### 3.2.1 Background

Fig. 3.2 illustrates an example of recent work in Penetrative AI [71], where various sensor data are textualized and embedded in a prompt, which is further used to instruct the LLM to perform inference tasks on the textualized sensor data.

Preliminary results indicate that the common knowledge embedded in LLMs can be leveraged to accomplish various real-world AIoT tasks (*e.g.*, inferring a user’s location via WiFi fingerprints, and counting heartbeats via raw ECG data). However, the inference processes of LLMs are largely opaque to users, rendering the results less explainable and trustworthy. Moreover, raw sensor data is transmitted from

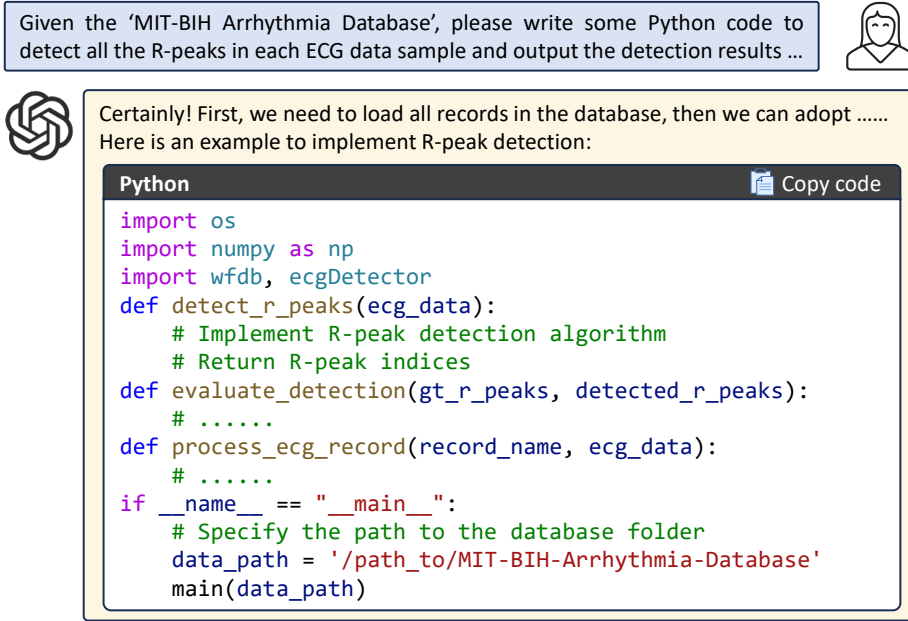


Figure 3.3: An example of direct code generation.

users to LLM servers, raising privacy concerns about sensor data. Furthermore, limited by token size, existing work down-samples and quantizes raw sensor data, leading to degraded inference performance. The remote processing at LLM servers also necessitates the round-trip transmission of prompts and results over the network, which increases response latency.

To overcome these issues, we explore a new approach that leverages LLMs to synthesize AIoT programs and executes the programs locally to process users' data. This new approach allows developers to examine and verify the synthesized programs, protect data privacy, process sensor data streams without compromising data resolution or quality, and avoid transmission time over networks.

### 3.2.2 Preliminary Experiments

The latest LLMs [88, 38, 89] have demonstrated extraordinary proficiency in generating code snippets. For example, Mercury [88] leverages LLMs to generate code for well-defined programming tasks. *Is it feasible to instruct LLMs to synthesize*

*programs that can tackle AIoT tasks?* Our preliminary results show that it is possible yet extremely challenging for LLMs to synthesize functionally correct programs for AIoT tasks. Taking heartbeat detection as an example, as depicted in Fig. 3.3. When we instruct the LLM to generate a program to process raw ECG waveform and detect heartbeats, the LLM can only generate a few null functions without concrete implementation or import some nonexistent packages.

We hypothesize that the reasons for this might be threefold: 1) LLMs lack domain-specific knowledge, let alone the latest algorithms tailored for AIoT tasks. As a result, for highly specialized AIoT applications, LLMs can only offer some suggestions or generate high-level code outlines rather than detailed functional implementation. 2) AIoT applications typically require systematic programming, where multiple functional modules are first developed for different subtasks (*e.g.*, signal preprocessing, data cleaning, neural network initialization), which are then constructively integrated to form a comprehensive and cohesive program. This development process involves much higher reasoning, planning, and programming complexity than other simple well-defined programming tasks. 3) Current LLMs inherently lack code validation and optimization mechanisms to ensure the correctness of synthesized programs and improve the performance of programs in terms of execution efficiency and inference accuracy in AIoT program synthesis tasks.

To validate the above hypotheses, we conduct follow-up experiments 1) by providing necessary domain-specific knowledge to facilitate the design and implementation of corresponding algorithms to address the AIoT task and 2) by explicitly instructing the LLM to synthesize programs with clear structures via a divide-and-conquer approach. Fig. 3.4 illustrates the code generation process involving user intervention. We first manually retrieve relevant background knowledge (*e.g.*, definitions of ECG data and R-peak, research papers about heartbeat detection) from the Internet and feed the information to the LLM, enabling in-context learning. Second, we

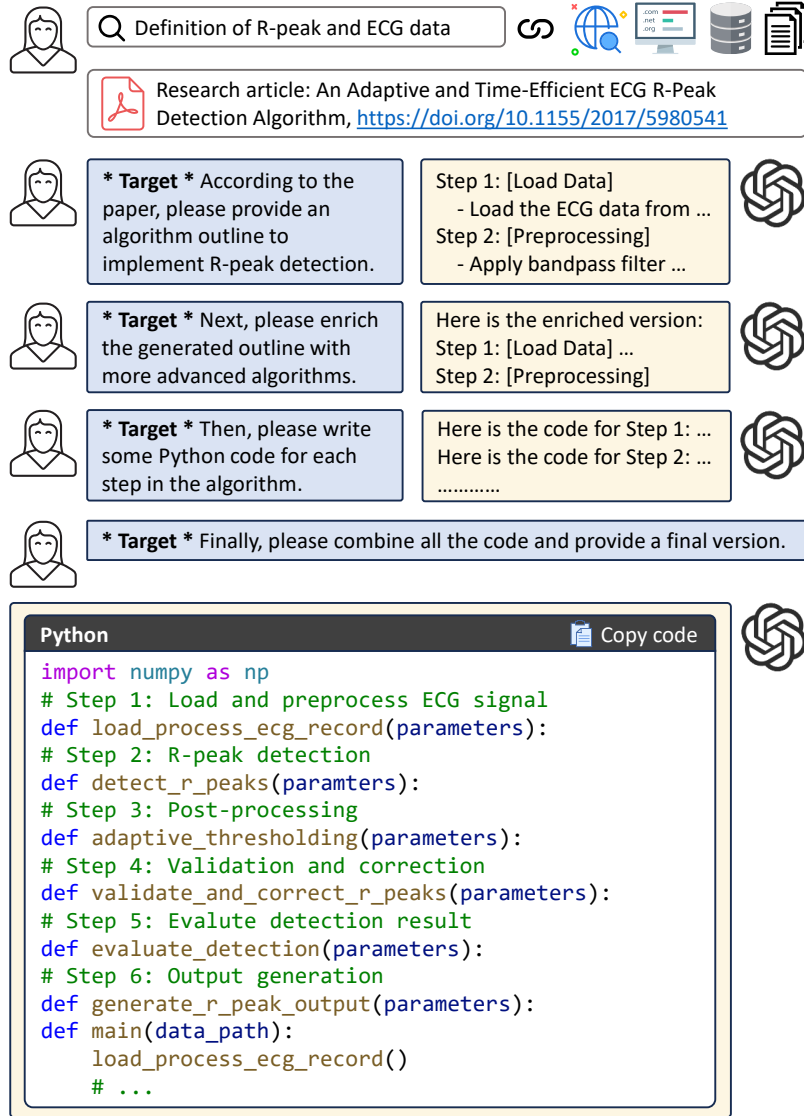


Figure 3.4: Code generation with user intervention.

instruct the LLM to learn the relevant context and comprehend the papers. Then, we ask the LLM to generate an outline of the algorithm in the paper. We further request the LLM to enrich the outline with more advanced and detailed technologies. Later, we ask the LLM to generate code snippets corresponding to each step of the algorithm. The final program is thereafter synthesized by integrating all the code snippets. Finally, we fix bugs if there are any, and execute the program to evaluate its performance with test data. We further give feedback and ask the LLM to refine

the program. With several rounds of iterations, the synthesized program evolves and improves its performance in the task.

In summary, although the synthesized programs eventually achieve reasonable performance in the tested AIoT tasks, this LLM-driven development method demands specialized domain expertise and constant manual intervention through several rounds of iterations for program optimization.

### 3.2.3 Motivation & Key Ideas

In this thesis, we aim to develop an LLM-driven automated natural language programming system named **AutoIOT** to synthesize programs for AIoT applications. **AutoIOT** features three key modules: 1) *Background knowledge retrieval* module that automatically collects domain knowledge from the Internet for in-context learning; 2) *Automated program synthesis* module that emulates the program development lifecycle [90] via CoT prompting. This module decomposes an AIoT task into several subtasks and generates corresponding functional code snippets; and 3) *Code improvement* module that executes the synthesized program and feeds the compiler and interpreter feedback to the LLM, facilitating iterative code correction and improvement. We note that although the program synthesis process needs communication and interaction with remote LLM servers, the synthesized program can be executed locally on the client side. This approach fundamentally differs from existing approaches such as Penetrative AI [71], and allows users to not only preserve data privacy but also improve the interpretability of synthesized programs as well as inference results.

## 3.3 System Overview

**AutoIOT** builds an intelligent agent that can automatically synthesize programs to fulfill user requirements in AIoT applications. As shown in Fig. 3.5, **AutoIOT** com-

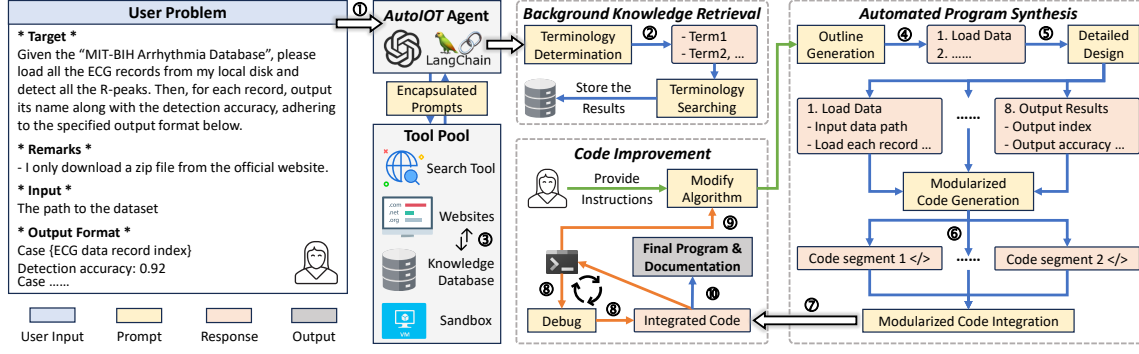


Figure 3.5: The system overview and workflow of AutoIOT.

prises three key modules: *background knowledge retrieval* (§ 3.4.2), *automated program synthesis* (§ 3.4.3), and *code improvement* (§ 3.4.4).

Users can specify their requirements on AIoT applications in natural language (①). Then, the *background knowledge retrieval* module identifies a set of relevant terminologies (②) and searches over the Internet (③). With the retrieved domain-specific knowledge, the *automated program synthesis* module instructs the agent to draft an algorithm outline (④). The agent is then requested to elaborate on each step of the algorithm and produce a detailed design (⑤). Such a process decomposes a complex AIoT task into several manageable subtasks. Then, the agent is instructed to generate a code segment for each subtask (⑥). Afterward, the agent is requested to integrate the codes for subtasks and synthesize the final program (⑦). Next, the *code self-improvement* module executes the synthesized program and feeds the compiler and interpreter output back to the agent. The agent iteratively corrects syntax and semantics errors (⑧). With the obtained output from the synthesized program, AutoIOT explicitly instructs the agent to explore more advanced algorithms using the web search tool, aiming to optimize the program and improve the performance of inference tasks (⑨). After several iterations (④-⑨), AutoIOT will present the final program with detailed documentation (⑩). In addition, AutoIOT provides an interface for users to offer specific algorithms or instructions for code improvement.

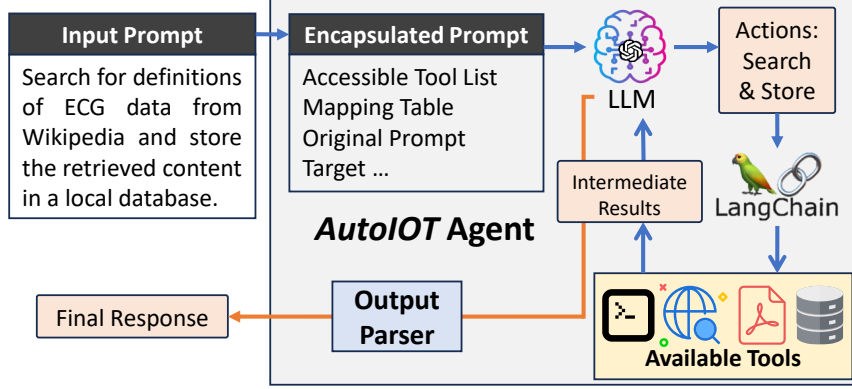


Figure 3.6: **AutoIOT** agent answers user’s prompt with LLMs and LangChain tools.

To enable the interaction between the LLM and the web search tool, the knowledge database, and the code executor, we leverage the LangChain [91] framework to build an intelligent agent. LangChain assembles various tools (abstracted as functions) and provides descriptions of the available functions (added into prompts) to the LLM. With such a prompt, the LLM performs reasoning and selects suitable tools to answer the user’s query (potentially via multiple rounds of function invocations and message exchanges initiated by the LLM). This approach allows the LLM to answer queries that require context information (*e.g.*, local weather, user’s local documents), augmenting the LLM with retrieved knowledge.

Taking step ③ in Fig. 3.5 as an example, Fig. 3.6 shows how **AutoIOT** agent works with LangChain tools and the LLM to answer the query about terminology searching via network. Given an input prompt, **AutoIOT** encapsulates it with additional information (*e.g.*, a list of available tools) and sends it to the LLM. Then, the LLM performs a sequence of actions possibly leveraging the available tools in the list, and generates an output. The output will then be passed to a parser to generate the final response. We note that the above processes involved in step ③ as well as other steps (①-⑩ in Fig. 3.5) are all automatically orchestrated by the **AutoIOT** agent.

**Usage Scenario.** Suppose a user wants to develop a heartbeat detection ap-

plication, she can interact with `AutoIOT` with natural language, which describes her requirements for the application. Then, `AutoIOT` will automatically synthesize a corresponding program and documentation for the user. Following the instructions in the documentation, she can deploy and execute the program on a target device, which contains the patient’s ECG data. The program will then generate the final heartbeat detection results for her.

## 3.4 System Design

In this section, we select heartbeat detection as an example to illustrate how `AutoIOT` works.

### 3.4.1 User Interface

Users can describe an AIoT task in natural language as input to `AutoIOT`. To help the LLM interpret the intention and desired outcome (*i.e.*, synthesized program and inference results), we design a prompt template for the user to describe the problem, since LLMs can comprehend and process well-structured instructions more efficiently [92]. As shown in Fig. 3.5, the user problem includes four parts: target, remarks, and program input and output specifications. The target part describes the user’s objective and task in natural language, *e.g.*, “Given the MIT-BIH Arrhythmia Database, please load all the ECG records and detect all the R-peaks. Then, evaluate the detection results and output the detection accuracy for each record.” The remarks provide additional information, *e.g.*, “I only downloaded a zip file from the dataset’s official website”. The program input and output specifications clarify the I/O format of the synthesized program. The path to the dataset is required during the *code improvement* process for program execution and optimization.

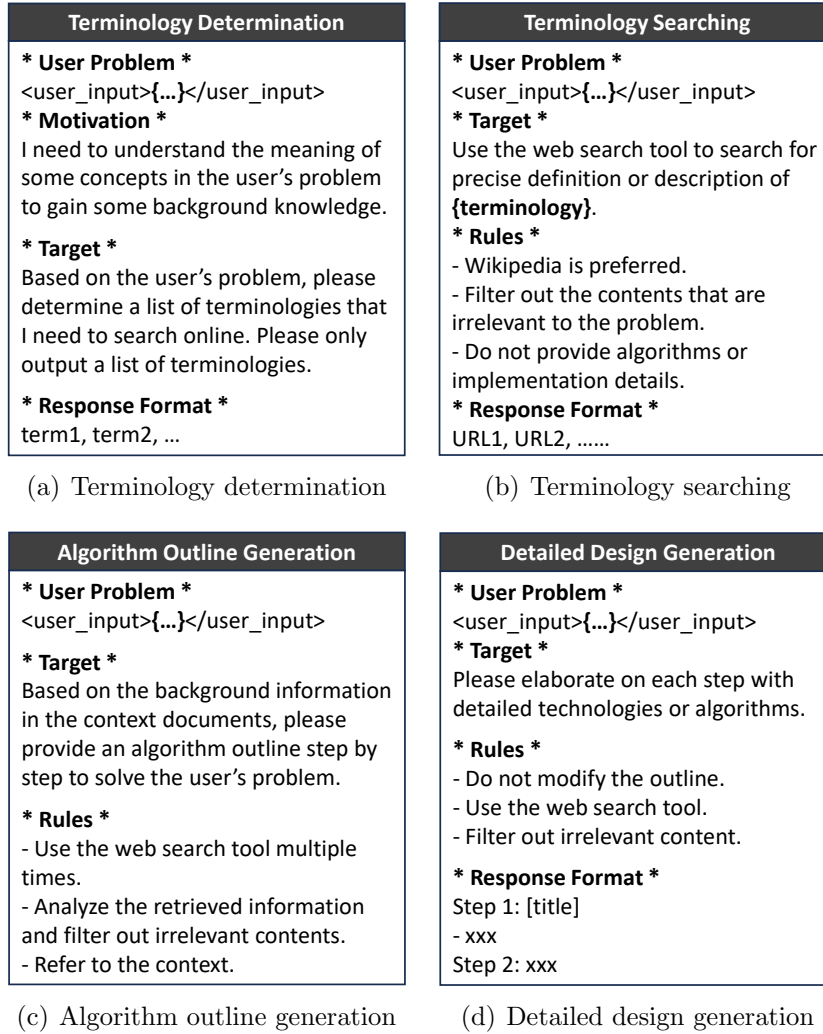


Figure 3.7: The prompt template for (a) terminology determination, (b) terminology searching, (c) algorithm outline generation, and (d) detailed design generation.

### 3.4.2 Background Knowledge Retrieval

LLMs are typically trained on extensive and pre-collected corpus datasets that include a wide range of general common knowledge over the Internet. These training datasets, however, may not include domain-specific knowledge or the latest advances in research literature. In the rapidly evolving AIoT field, with new technologies and algorithms constantly emerging, the knowledge gap is particularly pronounced. To fill this gap, we develop the *background knowledge retrieval* module to automatically

identify and fetch necessary information online so as to enable in-context learning for LLMs augmented with up-to-date domain knowledge.

**Terminology Determination.** The *background knowledge retrieval* module first instructs the agent to identify some relevant key terminologies given the user problem, with the prompt shown in Fig. 3.7(a). For example, given the user problem in Fig. 3.5, the terminologies generated by the LLM are “MIT-BIH Arrhythmia Database”, “ECG data”, and “R-peaks”. Next, to obtain the relevant knowledge, AutoIOT instructs (Fig. 3.7(b)) the LLM to actively search for the definitions and descriptions of these terminologies from public websites, such as Wikipedia and GitHub, with the web search tool. Additionally, for the terminologies with multiple interpretations, AutoIOT requests the LLM to filter out the irrelevant content and focus on those pertinent to the user problem.

**Context Database Construction.** After retrieving necessary information from relevant websites, AutoIOT uses OpenAI’s text embedding [93] to convert the HTML documents into vector representations, containing semantic meanings comprehensible to the LLM. These representations are then used to build a local vector database with Faiss [94], serving as a contextual knowledge base for the LLM. During the inference process, the LLM retrieves relevant content from the database with a high degree of similarity to the user problem in the vector space. This approach ensures that the LLM understands the user problem and objective with necessary context information and domain knowledge.

**Remarks.** The *background knowledge retrieval* module is user-friendly, as it is operated automatically by our AutoIOT agent without any user intervention. In addition, we provide an interface for users to explicitly complement necessary background information as well as highly specialized domain knowledge (*e.g.*, research papers, detailed algorithm descriptions) to enrich the context database.

### 3.4.3 Automated Program Synthesis

As observed in our preliminary experiments (§ 3.2.2), if we directly instruct LLMs to generate programs for AIoT applications, multiple subtasks should be undertaken manually by the user. Specifically, the user needs to decompose the AIoT task into several subtasks and request the LLM to generate a solution for each subtask. After integrating the solutions, the user has to manually debug, execute, and improve the program. Although the synthesized program eventually meets the user’s requirement, the program synthesis necessitates frequent user intervention and active involvement throughout the process, which is cumbersome and time-consuming. To address this problem, we develop the *automated program synthesis* module, aiming to automate the programming procedure, reduce the involved workload, and improve the development efficiency and user experience. In particular, the *automated program synthesis* module uses Chain-of-Thought (CoT) prompts to guide LLMs through step-by-step reasoning processes, thereby enhancing their capability of tackling complex AIoT problems by mimicking human-like divide-and-conquer reasoning processes.

**CoT 1: Algorithm Outline Generation.** AutoIOT first prompts the LLM to analyze the user problem and design a preliminary algorithm outline. As illustrated in Fig. 3.7(c), the prompt for algorithm outline generation consists of three parts: 1) The “User Problem” is reiterated at the beginning to ensure continuity and coherence in the LLM’s responses since the LLM may forget the previous context [95]. 2) The “Target” specifies our request, *i.e.*, algorithm outline generation. 3) The “Rules” add detailed instructions for quality assurance. For example, AutoIOT explicitly requests the LLM to actively search for advanced AIoT algorithms using the web search tool throughout the process. AutoIOT also asks the LLM to filter out irrelevant information. With such well-structured prompts, the LLM can generate an algorithm outline according to the problem specification.

**CoT 2: Detailed Design Generation.** Given the generated algorithm outline, the LLM is then tasked with further elaborating on each step in the outline with more detailed technologies and algorithms to refine the approach comprehensively. The prompt for this stage includes “User Problem”, “Target”, and “Rules”. In addition, it also includes a new requirement - “Response Format” to specify the expected format of the LLM’s output, as illustrated in Fig. 3.7(d). Given such a prompt, the LLM can generate detailed steps and specific actions in each step to achieve the overall objective and solve the user problem. In this stage, essentially **AutoIOT** guides the LLM to decompose the AIoT task into multiple subtasks, facilitating a divide-and-conquer strategy to synthesize the corresponding program in the next stage.

**CoT 3: Modularized Code Generation.** Given a set of subtasks generated at the previous stage, **AutoIOT** instructs the LLM to generate one function for each module with a clear function name, signature, and input/output specification. The modules can then be developed independently, ensuring specific functionalities are effectively implemented according to the algorithm descriptions generated above. This divide-and-conquer approach is proven effective in synthesizing complex programs by fully exploiting the modularized code generation capabilities [86] of LLMs.

In our initial trials, we observed that LLMs sometimes generate null functions with placeholders, invoke undefined functions, or import nonexistent packages. To tackle this problem, **AutoIOT** adds more stringent rules and requirements to explicitly ask the LLM to avoid generating null functions and verify the availability of imported packages and invoked functions by involving the web search tool. With the prompts shown in Fig. 3.8(a), the LLM can generate cohesive code segments with detailed comments for each module, facilitating the module integration in the next stage.

**CoT 4: Modularized Code Integration.** Given the generated code segments, **AutoIOT** prompts the LLM to constructively integrate all modularized code segments

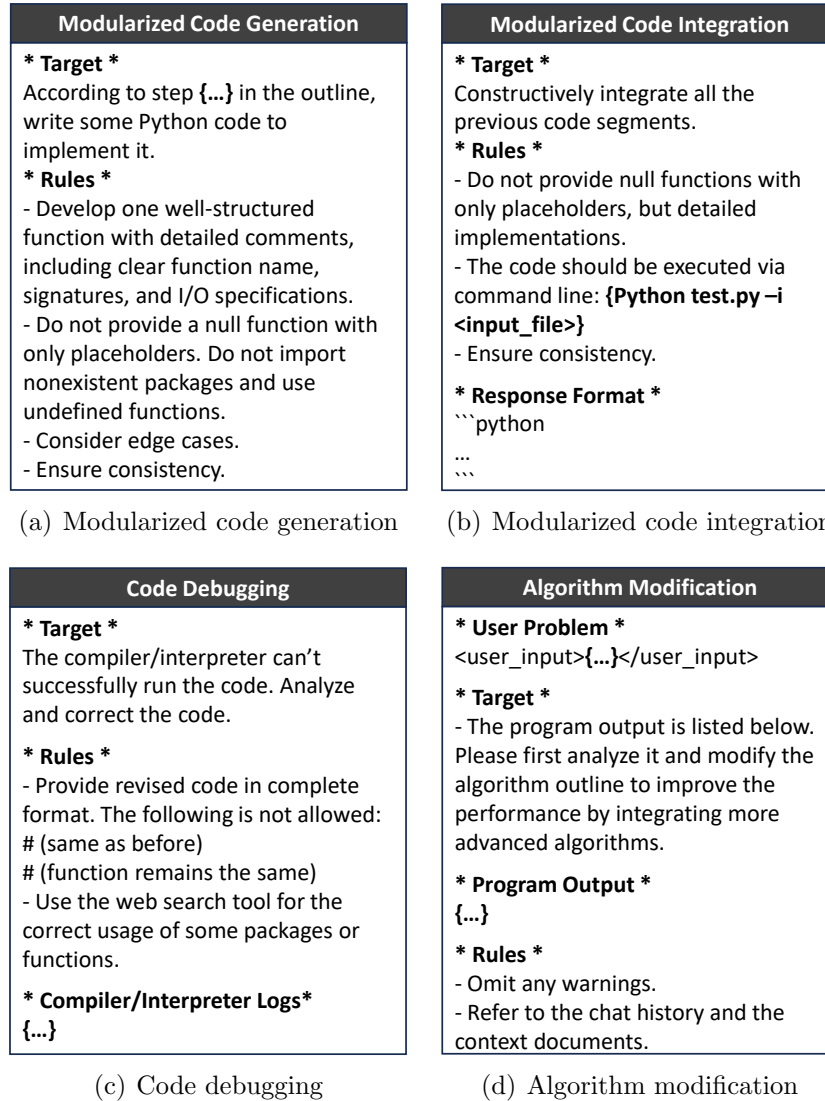


Figure 3.8: The prompt template for (a) modularized code generation, (b) modularized code integration, (c) code debugging, and (d) iterative code improvement.

and create a cohesive and comprehensive program. Since the code generated for different modules may have disparate input/output variable names, **AutoIOT** first prompts the LLM to ensure the consistency among all the modules and synthesize the final program without null functions as illustrated in Fig. 3.8(b).

For the convenience of code execution, debugging, and optimization, **AutoIOT** asks the LLM to add a main function so that the program can be directly executed

from the command console (*e.g.*, `python3 test.py -i <input_file>`). Moreover, `AutoIOT` also asks the LLM to generate user documentation in Markdown format, specifying how to properly install, execute, and troubleshoot the program for end users.

**Remarks.** The *automated program synthesis* module facilitates a seamless transformation from natural language to a readily executable program with CoT prompts. `AutoIOT` is regarded as an experienced developer, adept at decomposing complex AIoT tasks into multiple modules, generating modularized code, and organically integrating them. The program synthesis process can be fully automated by the agent.

### 3.4.4 Code Improvement

In § 3.2.2, we found that the LLM can evaluate and improve the code with heavy user intervention. To alleviate the user’s workload involved in debugging and code optimization, we develop the *code improvement* module.

**Automated Debugging.** Upon obtaining the final program after integration, `AutoIOT` constructs a code executor to run the generated code within a virtual environment (*e.g.*, a sandbox), ensuring safe and controlled code execution. The code executor loads the sensor dataset from the user’s local device for program execution and exports the compiler or interpreter output to the LLM. If the program encounters execution issues (*e.g.*, syntax or I/O errors), `AutoIOT` embeds the logs from the compiler into a prompt (Fig. 3.8(c)) and instructs the LLM to debug the code for several rounds of interactions until the generated code can be executed successfully.

**Code Optimization via Algorithm Modification.** To achieve better performance, `AutoIOT` progressively refines the synthesized program via several iterations. In particular, `AutoIOT` processes the test dataset with the first version of the integrated program. Then, `AutoIOT` prompts (Fig. 3.8(d)) the LLM with the context

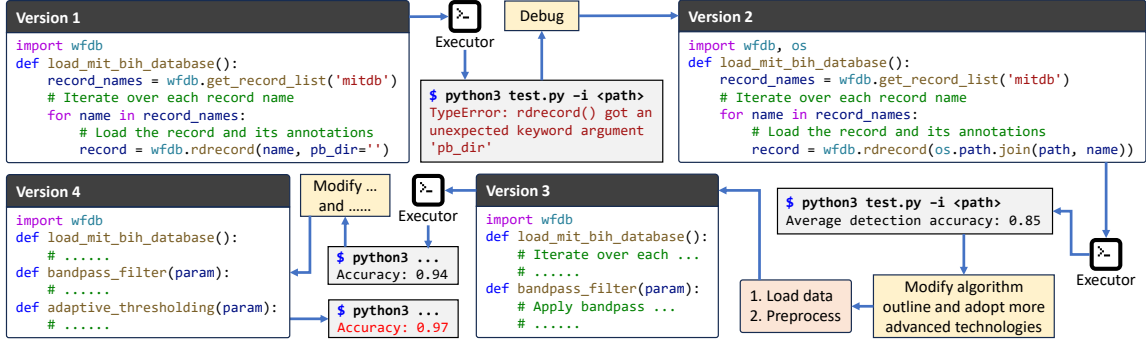


Figure 3.9: An example of code improvement via iterations (details omitted).

information (*e.g.*, algorithm outline, chat history) of generating the first program as well as the program output, and asks the LLM to improve the performance by integrating more advanced algorithms. Specifically, **AutoIOT** uses the web search tool to search for solutions that can achieve higher accuracies referring to academic papers and relevant webpages. This initiates a new recursive cycle of program synthesis, starting from refining the algorithm outline accordingly, enriching the outline with the retrieved algorithms, generating modularized code for the updated design, combining the modularized code, debugging, and improving code quality. This code optimization cycle is not a one-time process but is repeated multiple times. Empirically, **AutoIOT** takes five iterations to progressively generate five different programs, striking a balance between thoroughness and efficiency. Finally, **AutoIOT** requests the LLM to analyze the execution results of all the programs and select the one that achieves the best performance as the final program.

Fig. 3.9 shows a specific example of how the synthesized program evolves over multiple iterations. Version 1 is generated with a bug. By providing the error message to the LLM, **AutoIOT** can automatically fix the bug and generate Version 2. **AutoIOT** further instructs the LLM to modify the algorithm and adopt advanced technologies to generate new versions that iteratively achieve higher accuracies.

**Supporting User-in-the-Loop Optimization.** After each iteration of code op-

timization, `AutoIOT` also provides an interface for the user to *optionally* provide instructions that can help the LLM improve the synthesized program. For example, when the user finds that the LLM fails to recall relevant information from the retrieved contents, the user can prompt `AutoIOT` to refer to a specified algorithm or provide a recent academic paper. This enables a user-in-the-loop optimization that requires minimal user intervention and promotes code optimization iteratively.

**Remarks.** The *code improvement* module automates code debugging and optimization by leveraging the LLM’s proficiency in debugging and refining code based on the compiler and interpreter feedback [96, 97]. This automation not only releases the manual burden but also heralds a new era where AIoT applications can evolve iteratively and autonomously with minimum user intervention.

## 3.5 Experiment Setup

### 3.5.1 Implementation

We implement `AutoIOT` with GPT-4 [38] based on LangChain [91], which provides various tools (*e.g.*, web search engine, vector database, *etc*) for LLMs to collect relevant information. We select Tavily [98] as the web search tool to search for relevant information. It uses OpenAI’s text embedding model [93] to convert the retrieved webpages into vector representations. `AutoIOT` then uses Faiss [94] for efficient similarity search of vector representations. The code executor controlled by `AutoIOT` is deployed on a Linux Ubuntu workstation equipped with an NVIDIA RTX 4090 GPU.

### 3.5.2 AIoT Applications & Datasets

We select four representative AIoT applications from the domains of healthcare and human activity recognition (HAR). Unlike other program synthesis tasks, these AIoT

tasks require domain-specific knowledge and highly specialized algorithms in signal processing and machine learning.

**Heartbeat Detection.** R-peak detection in electrocardiogram (ECG) data is a crucial task in cardiac signal processing, serving as a foundational step for heart rate variability studies, and arrhythmia detection [99]. We use MIT-BIH Arrhythmia Database [100] and five representative baseline algorithms, including Hamilton [101], Christov [102], Engzee [103], Pan-Tompkins [104], and SWT [105].

**IMU-based Human Activity Recognition.** Inertial measurement unit (IMU)-based HAR enables continuous identification of a wide range of daily activities (*e.g.*, sitting, walking) by capturing and analyzing motion characteristics from the IMU data [106, 107]. For the baselines, we select five open-source GitHub repositories: LSTM-RNN [108], 1D-CNN [109], Conv-LSTM [110], BiLSTM [111], and NN [112]. We compare their performance with AutoIOT on the WISDM dataset [113].

**mmWave-based Human Activity Recognition.** mmWave can capture fine-grained human gestures with high resolution [114, 18, 19]. We select the XRF55 dataset [115] with the models proposed in the paper as the baselines, including ResNet-18, 34, 50, 101, and 152. This task is more challenging because: 1) The XRF55 dataset was recently published on websites only a few months ago, which means LLMs have not yet seen knowledge about this dataset; 2) The mmWave data has high dimensionality, necessitating the use of more sophisticated models with optimized configurations.

**Multimodal Human Activity Recognition.** By leveraging different sensors to capture complementary information, HAR systems can achieve higher robustness and versatility. We select the Harmony dataset [69] containing three sensor modalities: audio, depth image, and radar. The baseline system consists of three encoders, each designed to extract unique features from the respective modalities, followed by feature concatenation and a classifier model. This task is also challenging for

AutoIOT as it involves the fusion of data from different modalities with cross-modal interaction.

## 3.6 Evaluation

### 3.6.1 Metrics

We adopt the following evaluation metrics: 1) *Task accuracy*: we repeat the experiment 10 times and calculate the average task accuracy. In heartbeat detection, we consider task accuracy as the percentage of correctly identified peaks within a predefined tolerance window compared to the ground truth. In HAR, we consider classification accuracy. 2) *MAE*: We use medium absolute error (MAE) to measure the discrepancy in beat positions between the predicted R-peaks and the ground truth. 3) *Communication cost*: we use *psutil* [116] to monitor the network traffic. 4) *Wall-clock execution time*: we record the total time consumed from the moment the user inputs the problem to the generation of the final inference results for all the sensor data. 5) *Memory consumption*: we record the GPU memory consumption during code execution if AI models are adopted. 6) *Inference time per sample*: we compute the inference time per data sample if AI models are used.

### 3.6.2 Performance against Baselines

#### Average accuracy & MAE

Fig. 3.10(a) shows the heartbeat detection accuracy with MAE of AutoIOT (denoted as A.) and baselines. First of all, AutoIOT can synthesize a program automatically to achieve comparable performance with baselines in the heartbeat detection task. In addition, the standard deviations of the AutoIOT-synthesized programs are low, achieving 2% and 0.15 in terms of accuracy and MAE, respectively. More surprisingly, the automatically synthesized program can even beat some of the baselines!

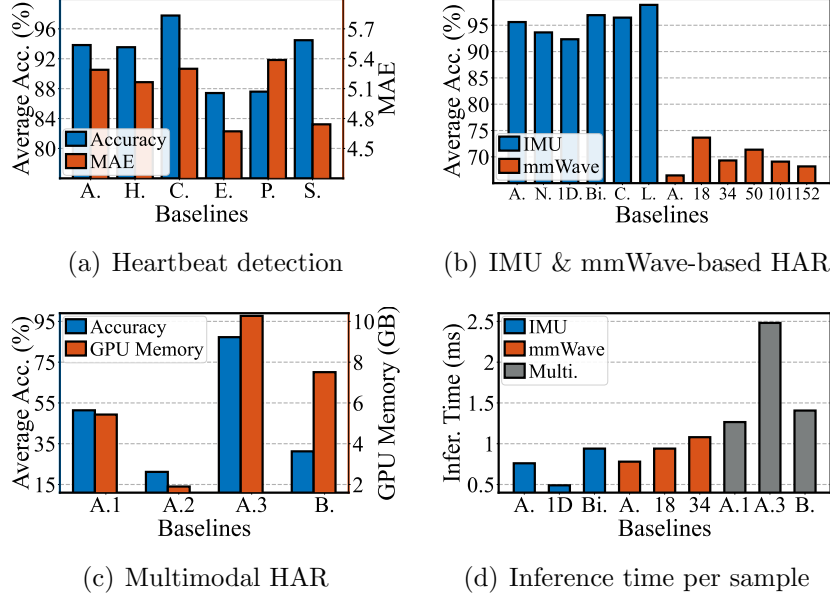


Figure 3.10: The overall performance of the four IoT applications. In (a), A. for AutoIoT, H. for Hamilton, C. for Christov, E. for Engzee, P. for Pan-Tompkins, and S. for SWT. In (b), N. for NN, 1D for 1D-CNN, Bi. for BiLSTM, C. for Conv-LSTM, L. for LSTM-RNN, and  $n$  for ResNet- $n$ . In (c) & (d), A.1, A.2, and A.3 for three different AutoIoT-generated programs; B. for the baseline in the multimodal HAR application.

For example, the synthesized program achieves higher detection accuracy than Pan-Tompkins (P.) and Engzee (E.). Moreover, it yields a lower error rate than Christov (C.) and Pan-Tompkins (P.). To investigate the reasons behind this, we examine and analyze the synthesized program.

**We learned:** 1) Armed with the web search tool, the synthesized program implemented a few basic as well as sophisticated signal processing methods, including bandpass filtering and stationary wavelet transformation in preprocessing, and adaptive thresholding in postprocessing. Some selected algorithms are well-known and widely adopted, while others are less likely to be chosen, even by experienced programmers with domain expertise. Unlike narrowly focused, well-defined simple programming tasks, AIoT tasks typically require systematic integration of multiple algorithms and components to achieve optimal system performance, which creates

opportunities for **AutoIOT** to explore more possibilities in automatically synthesizing optimized programs that could outperform not all but some representative baselines.

2) Given a single performance objective, we notice that the LLM carries out extensive optimization, sometimes at the cost of other equally important metrics. For example, when instructed to improve the detection accuracy, the synthesized program sets a larger tolerance window, which increases the chance of correctly detecting heartbeats (true positives) at the cost of increased false positives. Considering AIoT applications' complexity and multiple competing or even contradicting objectives, user requirement specification needs to be as complete and comprehensive as possible, which necessitates domain expertise and system development experience.

3) We found that the webpages returned by the web search tools are often about general algorithms due to their popularity and higher page rankings. Such popular algorithms, however, may not perform the best in domain-specific tasks. With minimum user intervention by providing specialized algorithms, **AutoIOT** can synthesize programs accordingly and achieve comparable performance to the baselines.

Fig. 3.10(b) shows the classification accuracy in two HAR tasks. We observe that **AutoIOT** outperforms NN and 1D-CNN while underperforms BiLSTM, Conv-LSTM and LSTM-RNN. The main reasons are twofold: 1) HAR tasks require both signal processing and machine learning algorithms, increasing the programming complexity to some extent; 2) Training neural networks requires fine-tuning of a vast array of hyper-parameters (*e.g.*, network architecture configurations, epoch number, learning rate, optimizer, and loss function). This significantly amplifies the instability of the generated code and calls for careful fine-tuning to achieve the best performance in practice. As a result, **AutoIOT** surpasses those baselines adopting simple model architectures (NN and one-dimensional CNN) but falls short against baselines using sophisticated architectures (BiLSTM and Conv-LSTM) with highly optimized hyper-parameters. Although during code improvement, some synthesized programs define

a set of configurations and adopt a searching strategy to obtain optimal hyper-parameters, the performance still remains slightly lower than some baselines. This is because the determination of the optimal configurations for machine learning models is typically a trial-and-error process, requiring substantial human effort. Fortunately, we observe that if the user provides a potential search space in advance, the LLM can design a search algorithm to try different hyper-parameter configurations and select the one with the best performance.

For multimodal HAR, the input instruction (A1) includes the basic information of the task, *i.e.*, the task target, the dataset specifications, and the output format. Based on that, we create two additional variations: one with a GPU-memory-constrained requirement (A2) and another with a high accuracy requirement (A3). We then feed the instructions into **AutoIOT** and measure the accuracy and inference time of the synthesized programs, with results shown in Fig. 3.10(c). By analyzing the three synthesized different programs, we observe that: 1) All the synthesized programs adopt a similar workflow as the baseline system, *i.e.*, they first construct three encoders to extract effective features from the three modalities, then concatenate these features and feed them into a classifier for activity recognition. This implies that benefiting from our CoT-based problem-solving paradigm, **AutoIOT** recognizes the workflow and architecture as effective and standard for handling multimodal data-related tasks, which is consistent with most of the existing methods [69]. 2) **AutoIOT** can adjust the generated code to fulfill different requirements. The second program consumes less memory than others due to the resource constraint requirement, resulting in lower accuracy but reduced inference time (Fig. 3.10(d)). On the other hand, the third program adopts a more complex and larger model architecture, requiring more GPU memory and incurring a longer inference time. Such differences validate the capabilities of **AutoIOT** in accurately understanding and processing natural language-based user requirements. These observations further demonstrate the

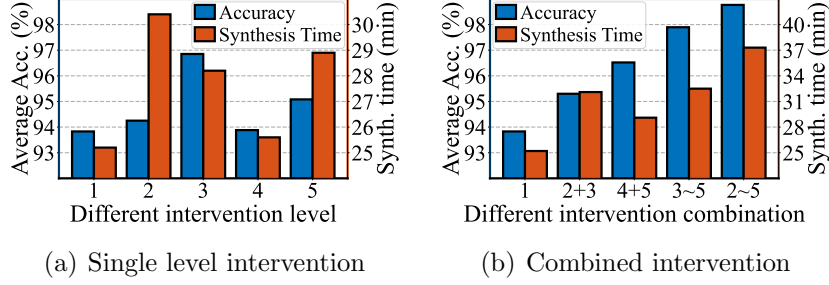


Figure 3.11: Different levels of user intervention.

effectiveness of **AutoIOT** in ensuring the correctness of user requirement understanding and the generated code, benefiting from our automatic self-improvement component.

### Communication cost. & wall-clock time

We select heartbeat detection as an example and measure the total communication cost with wall-clock execution time of **AutoIOT** and direct LLM inference as done in Penetrative AI. Specifically, ECG data is first down-sampled and segmented into multiple windows and then embedded into the prompt for LLMs’ inference. Experiment results show that **AutoIOT** requires 8MB of network traffic mainly for prompt transmissions, while Penetrative AI consumes more than 50MB mainly for sensor data transmissions. Besides, **AutoIOT** takes 25 minutes to complete the task with a dramatic reduction in inference time compared to Penetrative AI, which needs to send and process all windowed signals with the remote LLM serving. Notably, the standard deviations of both communication cost and wall-clock time of **AutoIOT** are both more stable than the baselines, achieving 200 KB and 2 minutes, respectively.

### 3.6.3 Sensitivity Analysis

**Different levels of user intervention.** To show how *automated program synthesis* improves user experiences, we evaluate the **AutoIOT**’s performance under five differ-

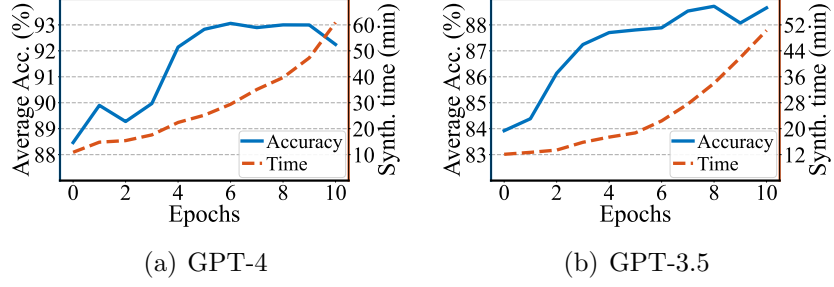


Figure 3.12: Different numbers of iteration.

ent levels of user intervention: 1) No intervention; 2) Intervention with user-provided domain knowledge; 3) Intervention with user-specified algorithms for program synthesis; 4) Intervention with user-based debugging; 5) Intervention with user-decided algorithm modification for code improvement. Fig. 3.11(a) shows the performance of AutoIOT with different levels of user intervention. When the user manually instructs the LLM to generate code according to specific hand-picked algorithms (*e.g.*, designed by experts or research papers), the average accuracy can be improved. This user-in-the-loop process becomes particularly advantageous when users have a higher level of domain expertise in AIoT, enabling them to design or select more advanced and robust algorithms. But it leads to increased synthesis time as the LLM has to revise outputs until the user is satisfied. Fig. 3.11(b) shows the performance with different user intervention combinations. We see that increased user involvement in the program synthesis process correlates with higher accuracy. However, this heightened engagement leads to significantly longer synthesis time and extra user overhead. Note that AutoIOT may not always be able to fix bugs and finish program synthesis tasks. In this case, user intervention with minimum effort is still a must. Thus, AutoIOT allows users to provide detailed instructions necessary for program synthesis by the LLMs.

**Different numbers of improvement iterations.** We vary the number of epochs for code optimization from 0 to 10 and evaluate the impact on the synthesized pro-

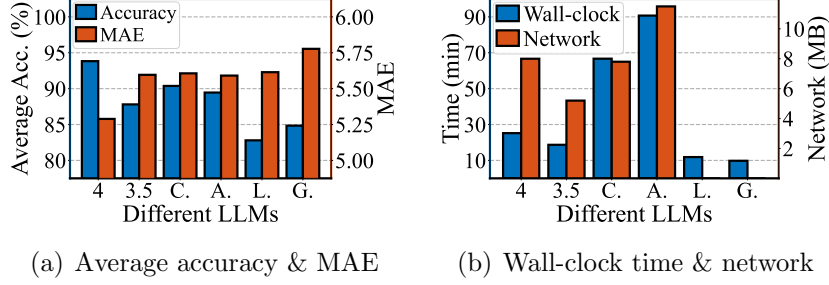


Figure 3.13: Different LLMs. (4 for GPT-4, 3.5 for GPT-3.5, C. for Cohere, A. for Anthropic Claude 2, L. for Llama2-7b, and G. for Gemma-7b.)

grams. As shown in Fig. 3.12, with more improvement epochs, the accuracies/synthesis time of the GPT-3.5/-4 generated programs gradually increase. However, after around 5 epochs, the marginal gain of average accuracy starts to diminish while the synthesis time increases dramatically. This is because, with longer conversation history, LLMs may fail to recall past context information and tend to generate inconsistent responses [95]. Therefore, we empirically set the epoch number to five.

**Impact of Different LLMs.** We select the following LLMs for comparison: GPT-4 [38], GPT-3.5 [117], Llama2-7b [118], Cohere [119], Claude 2 [120], and Gemma-7b [121]. Llama2 and Gemma are locally deployed in our lab. We select R-peak detection as an example with the Christov algorithm as the baseline. Fig. 3.13(a) shows that GPT-4 performs the best. Given the knowledge retrieved by the same tools, LLMs still need strong language understanding and reasoning capabilities to comprehend AIoT tasks and synthesize programs. Experiment results indicate that GPT-4 might have superior performance in language understanding and reasoning capability for this specific task. Although Llama2-7b and Gemma-7b achieve relatively lower accuracy, these two local models offer much faster response speeds.

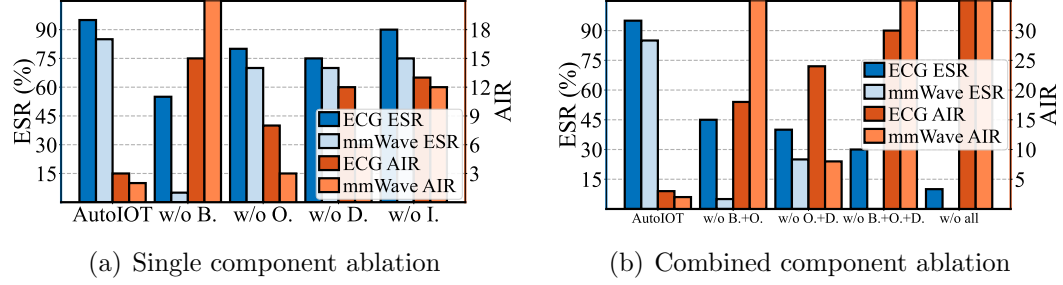


Figure 3.14: Ablation study. (B – background knowledge retrieval, O. – algorithm outline generation, D. – detailed design generation, I. – code improvement.)

### 3.6.4 Ablation Study

In the ablation study, we use two metrics to evaluate the code quality: 1) *Execution success rate* (ESR): the proportion of the code that can be executed successfully for the first time. 2) *Average iteration round* (AIR): the average number of improvement iterations required to achieve 80% accuracy.

**Background knowledge retrieval.** To explore the influence of the *background knowledge retrieval* module, we disable the web search tool and the knowledge database. We then instruct the LLM to synthesize 20 different programs with no user intervention. Fig. 3.14(a) shows the ESR and AIR of two AIoT tasks. We see that the ESR of heartbeat detection drops slightly while the ESR of mmWave-based HAR exhibits a significant drop. This is because the mmWave-based HAR uses a newly published dataset, which has not been seen by the LLM. Therefore, the LLM does not know the dimensionality of the dataset and only randomly configures the hyperparameters of the neural network. Additionally, both applications require larger numbers of iterations to improve the accuracy of synthesized programs. We note that mmWave-based HAR even fails to achieve the expected accuracy (thus, its AIR is marked as infinite). A similar phenomenon is also observed in Fig. 3.14(b). The experiment results indicate that the background knowledge retrieval module plays a pivotal role for the LLM to retrieve up-to-date domain knowledge to augment the

program synthesis process.

**Chain-of-thought.** We evaluate the contribution of the algorithm outline generation step and the detailed design generation step during the CoT prompting, respectively. As shown in Fig. 3.14(a), when only one step is enabled, we observe a slight drop in ESR and an increase in AIR for both applications. When we disable both steps, the ESR drops significantly as shown in Fig. 3.14(b). Without the explicit guidance specified in the two steps, the LLM cannot synthesize executable programs and presents null functions with placeholders. Therefore, the CoT method with detailed instructions emulating the software development lifecycle plays a crucial role in helping LLMs synthesize executable programs.

**Code improvement.** The *code improvement* module contains automated debugging and code optimization. Since automated debugging is essential to ensure the executability of synthesized programs, we only conduct an ablation study on the code optimization step. With the program after the debugging step, we evaluate the *code improvement* module by directly instructing the LLM to modify the program over several iterations without providing the compiler or interpreter feedback. As shown in Fig. 3.14(a), without the *code improvement* module, the ESR is almost unaffected while the AIR increases significantly. This is because generating an executable program with no syntax errors is seldom influenced by this module. However, without feedback from the compiler or interpreter, more iterations are required since the LLM does not know which step of the algorithm should be modified or improved. Besides, due to the laziness of LLMs [122], synthesized programs tend to adopt simple and popular algorithms. This necessitates the *code improvement* module, which progressively directs the LLM to explore more advanced algorithms to improve the synthesized program.

Table 3.1: Participant information of user study (N=20)

| Category         | Background Information                                          |
|------------------|-----------------------------------------------------------------|
| <b>Gender</b>    | Female (45%), Male (40%), Prefer not to say (15%)               |
| <b>Age</b>       | Under 18 (10%), 18-30 (75%), 30-39 (10%), 40 and older (5%)     |
| <b>Education</b> | Bachelor (20%), Master (15%), Doctoral (60%), Others (5%)       |
| <b>English</b>   | Beginner (5%), Intermediate (25%), Advanced (60%), Fluent (10%) |
| <b>Expertise</b> | Expert (25%), Non-expert (75%)                                  |

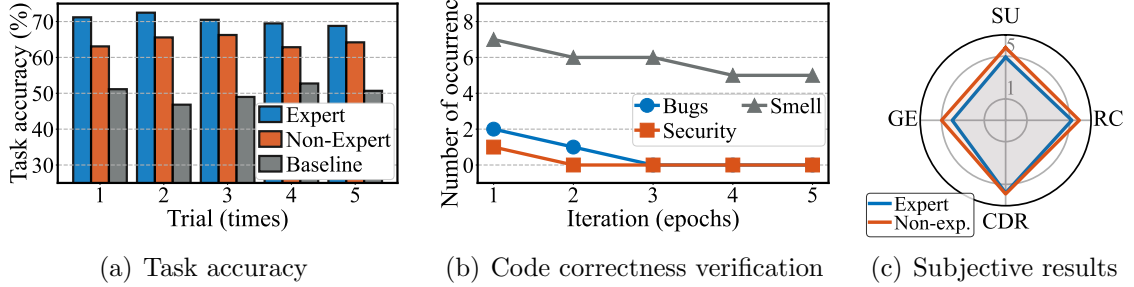


Figure 3.15: User study (a &amp; b: objective evaluation; c: subjective evaluation)

### 3.6.5 User Study

To investigate the utilities of AutoIoT, we conduct a user study (N=20) by inviting 5 expert and 15 non-expert users, whose detailed background information is listed in Table 3.1. The expert users are PhD students and professors with work or research experience in the IoT field and have developed many IoT applications. We select human activity recognition using RFID data (XRF55 dataset [115]) as the IoT application, where a 1D Conv-based ResNet18 is the baseline.

**Objective Evaluation.** We first repeatedly measure the average task accuracy (classification accuracy) after executing the synthesized programs, with the results shown in Fig. 3.15(a). We see that the programs synthesized by the two groups of users outperform the baseline across multiple trials. Besides, the programs synthesized for experts typically perform better than those for non-experts. The main reason is that experts tend to provide more information in the specifications (*e.g.*, the dataset format, the training workflow). Consequently, AutoIoT can synthesize

programs with more advanced algorithms and detailed specifications for expert users. Next, we use SonarQube [123] to verify the correctness of the generated code, including bug/logic errors, security issues, and code smells [124] after every improvement iteration. Code smells are not bugs but bad coding styles (*e.g.*, variable name mismatching regular expression) or potential weaknesses (*e.g.*, package version incompatibility). From Fig 3.15(b), we observe that several bugs and one security issue present initially are ultimately fixed by **AutoIOT**. **AutoIOT** may not be able to address all code smells, as they are closely related to coding styles [125]. Applying code smell correction to the retrieved algorithms and programs can be promising for **AutoIOT** to iteratively detect and fix code smell-related issues.

**Subjective Measurement.** We ask the users to execute the synthesized programs and rate **AutoIOT** based on four subjective metrics: 1) *System Utility (SU)* measures the user’s overall satisfaction with **AutoIOT**’s performance; 2) *Requirement Coverage (RC)* evaluates how well the user requirements are fulfilled by **AutoIOT**; 3) *Code & Documentation Readability (CDR)* measures the clarity and structure of the code and documentation; 4) *Generation Efficiency (GE)* accesses how acceptable the waiting time is for synthesizing the final program. All the above metrics are rated by the users on a scale from 1 (not at all) to 6 (more than expected). As shown in Fig. 3.15(c), the average GE of both user groups reaches 4.5, indicating that the waiting time for **AutoIOT** to synthesize the final program is acceptable for most users. Additionally, we find that non-experts tend to give higher scores (SU, RC, and CDR) than experts. Further examination reveals that non-experts tend to under-specify their requirements. Surprisingly, LLMs can sometimes provide comprehensive responses to meet their requirements. This ability is rooted in LLMs’ extensive training on diverse datasets and retrieved online information, enabling them to infer and bridge the gaps with relevant information [126, 127].

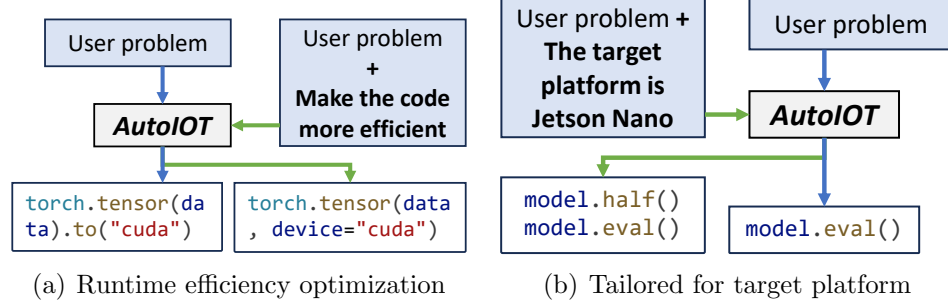


Figure 3.16: Further experiments.

### 3.7 Discussion

**Runtime Efficiency.** AutoIOT focuses on generating functionally correct code for IoT data processing. While the runtime efficiency of the programs is not optimized, the users can specify the extra requirements in the prompts. Fig. 3.16(a) shows an example that, given the user specification, the generated code adopts CUDA optimization and thereby achieves similar runtime efficiency compared to expert-optimized programs. This is because AutoIOT can retrieve and learn from hand-crafted optimizations via various online sources.

**Workflow Generalizability.** Existing IoT applications can be primarily categorized into four types based on the specific stages of the IoT workflow they address: 1) data collection applications rely on sensors (*e.g.*, IMU and radar) to gather raw information from the environment; 2) data transmission applications enable seamless communication across IoT networks to cloud or edge systems; 3) data processing applications typically adopt advanced algorithms or AI models to analyze and process IoT data; 4) decision-making and actuation applications parse the processed data to perform task automation or actuator control. In this thesis, AutoIOT is designed primarily for IoT data processing tasks by offering end-to-end solutions with executable programs. The prompts designed in AutoIOT are not limited to specific data patterns (time-series or high-dimensional) and processing pipelines (sequentially or

parallelly). For other IoT application workflows, users can specify the requirements in the prompt. For example, users can first instruct **AutoIOT** to develop a WiFi data collection application. Then, the synthesized program can be deployed on WiFi-related devices to capture WiFi data. Next, users can further request **AutoIOT** to process and analyze the collected WiFi data to perform HAR or other tasks. Moreover, some IoT applications may require executing multiple programs simultaneously for cross-device communication or synchronization. To tackle such applications, a possible solution can be deploying multiple **AutoIOT** agents with advanced collaboration mechanisms for enhanced cross-device and cross-program interaction. We leave the comprehensive generalization to other types of IoT tasks beyond pipelined data processing for future work.

**Real-World Deployment.** We show that **AutoIOT** can synthesize functionally correct IoT programs in Python with data processing accuracy as the main performance metric. **AutoIOT** is not limited to specific target IoT platforms. For example, regarding those resource-constrained MCU-class devices [26, 128], various practical requirements (*e.g.*, power management and network protocols) of the IoT device can be factored into **AutoIOT**’s prompts for program synthesis. Fig. 3.16(b) shows an example where a user explicitly specifies Jetson Nano, which has limited GPU resources, as the target platform. **AutoIOT** will then adopt half-precision training and inference for the AI model to save GPU memory. Moreover, users can even provide **AutoIOT** with handbooks of some dedicated IoT devices for reference.

**Memorization Issue.** In **AutoIOT**, sometimes the synthesized programs’ performance cannot be improved, even after multiple improvement iterations. This is because LLMs may forget the previous context, resulting in inconsistent code improvement suggestions during iterations [129]. In such cases, our user-in-the-loop optimization allows users to optionally provide instructions that can help the LLM improve the synthesized programs. For instance, users can explicitly instruct the

LLM to use the Pan-Tompkins algorithm for ECG data processing. Additionally, we can further upgrade **AutoIOT** by adopting existing memorization enhancement methods, such as context compression via RAG [130] and iterative summarization [131]. By integrating these orthogonal approaches with **AutoIOT**, each self-improvement iteration can contribute more positively to the performance of the generated code with enhanced context consistency.

**Privacy Concerns.** In **AutoIOT**, only user requirements are transmitted to the cloud LLM for processing. We proactively instruct **AutoIOT** to treat user configuration data (*e.g.*, the local file path) as program input. As such, during local execution, users can directly input the private configurations in the console (§ 3.4.3), rather than pre-defining them in the prompts for cloud LLMs. To further mitigate privacy concerns, one possible solution can be deploying a local LLM to handle all code generation and debugging tasks [39, 5].

## 3.8 Related Work

**Integration of LLMs with AIoT.** Existing integration methods have two main types: 1) Prompt-based methods embed raw sensor data into tailored prompts and instruct LLMs to perform various AIoT tasks. HARGPT [73] and LLMSense [132] embed textualized sensor data into prompts to show the proficiency of LLMs in comprehending IoT sensor data. They require transmitting raw sensor data to LLM servers, suffering similar issues as Penetrative AI. 2) Fine-tuning-based methods retrain LLMs with labeled datasets containing sensor data inference examples. LLM4TS [133] fine-tunes LLMs using sensor data with labels for time-series data prediction. However, these works demand high compute and memory resources. In contrast, **AutoIOT** explores a new approach to automatically synthesizing programs for AIoT applications without extra overheads.

Table 3.2: AutoIOT vs SOTA code LLMs

| Type                  | Name             | Mod.<br>Gen. | Sys.<br>Gen. | RAG | Auto. | Auto Debug<br>& Impro. |
|-----------------------|------------------|--------------|--------------|-----|-------|------------------------|
| <b>Code<br/>Gen.</b>  | DeepSeek         | ✓            | ✗            | ✗   | ✗     | ✗                      |
|                       | Coder [134]      |              |              |     |       |                        |
|                       | CodeLlama [49]   | ✓            | ✗            | ✗   | ✗     | ✗                      |
|                       | WizardCoder [85] | ✓            | ✗            | ✗   | ✗     | ✗                      |
| <b>Task<br/>Auto.</b> | AutoGPT [137]    | ✓            | ✓            | ✗   | ✓     | ✗                      |
|                       | MetaGPT [52]     | ✓            | ✓            | ✗   | ✓     | ✗                      |
|                       | AutoIOT          | ✓            | ✓            | ✓   | ✓     | ✓                      |

**Code LLMs.** Recent advances in code LLMs [49, 134] have demonstrated the potential to revolutionize software development. They can produce functionally correct code across various programming languages and input compiler output into LLMs for debugging and program refining [135, 136]. However, they require high computing resources with carefully selected datasets (StarCoder [50] needs an 815GB dataset and 512 A100 80GB GPUs). Worse still, they are unaware of the latest advances in highly specialized domains and cannot generate comprehensive solutions for complex IoT tasks. **AutoIOT** draws strength from these works and addresses specific technical challenges in IoT program synthesis, which require ever-evolving domain knowledge that the above LLMs have not yet assimilated. Table 3.2 compares **AutoIOT** with SOTA code LLMs. LLM-based code generation synthesizes functionally correct programs for a well-defined module, whereas LLM-based task automation generates a complete automation chain from requirements to solutions. **AutoIOT** lies at the intersection of these two approaches, with the objective of code generation and task automation. Additionally, RAG and automated code improvement jointly improve the code generation process in **AutoIOT**.

### 3.9 Chapter Summary

We propose **AutoIOT**, an LLM-driven automated natural language programming system for AIoT applications. Our system features three novel technical modules: background knowledge retrieval, automated program synthesis, and code improvement, transforming natural language descriptions into executable programs. Our experiments demonstrate the competitive performance of **AutoIOT** in synthesizing programs for a variety of AIoT applications, with comparable performance in challenging AIoT tasks and sometimes outperforming some representative baselines. This showcases the strong potential of exploiting the embedded common knowledge of LLMs to evolve AIoT application development.



# Chapter 4

## Localizing Small Language Models for AIoT Applications

### 4.1 Introduction

Large language models (LLMs) [138, 139] are revolutionizing various aspects of embedded system development and mobile computing, *e.g.*, smartphone task automation [65], advanced virtual assistants [140], and even IoT data comprehension [71, 132, 141, 142]. Code LLMs (*e.g.*, WizardCoder [85] and CodeLlama [49]) stand out as promising tools designed to synthesize programs based on user requirements described in natural language. As illustrated in Fig. 4.1, the integration of programming tools with code LLMs significantly enhances software development by automating code completion, code generation, bug detection, documentation writing, *etc.*

While powerful and promising, when confronted with IoT applications [143, 76, 75, 12, 9, 15, 78, 144] that require special domain knowledge, existing code LLMs tend to simply provide general solutions with sub-optimal performance (§ 4.2.2). This is because they focus on general-purpose programming tasks [145] rather than being tailored to any particular domain. Moreover, IoT-related knowledge and programs only occupy a small proportion of the datasets which code LLMs were trained

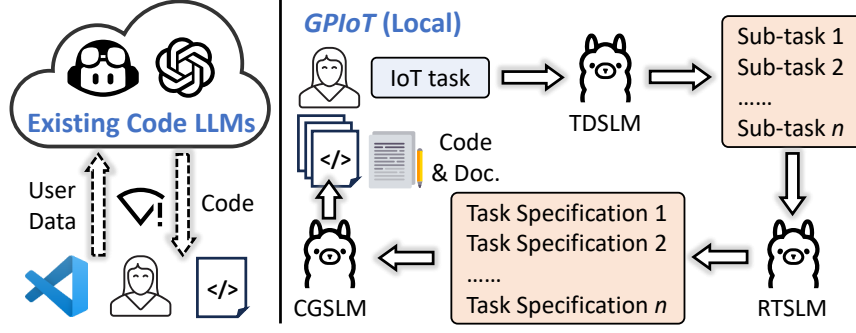


Figure 4.1: Existing code LLMs need to transmit sensitive data to remote servers. In contrast, GPIoT features three local SLMs to protect user privacy and reduce query costs.

on [146]. Consequently, IoT terminologies will be assigned a lower priority during inference with the generated code less dedicated to the IoT domain (§ 4.2.2). This motivates the following research question: *Can we build a code LLM specially tailored for IoT application code generation tasks?* If yes, we can synthesize IoT-related programs with higher task accuracy and efficiency, offering significant convenience for IoT developers.

A potential approach can be Retrieval-Augmented Generation (RAG) [147], which provides LLMs with retrieved domain knowledge to enhance their abilities in generating accurate and contextually relevant solutions. Existing works [4, 2, 148] construct a sophisticated LLM+RAG agent to gradually generate code through multiple intermediate steps via prompts. Nonetheless, they suffer from three main problems. 1) A powerful LLM with strong language comprehension capability is needed to learn from the retrieved knowledge. However, cloud LLMs (*e.g.*, GPT-4 [38]) may suffer from bad network conditions, high costs, and privacy concerns, while local LLMs (*e.g.*, Llama2-70b [118]) have harsh requirements in system resources (*e.g.*, memory and network). 2) Complicated RAG designs (*e.g.*, iterative retrieval [149]) are mandatory to ensure the correctness and high relevance of the retrieved knowledge, with extended processing time. Otherwise, LLMs may fail to focus on the IoT

context and still provide general solutions [150]. 3) Meticulously designed prompts are required to ensure that outputs must strictly follow pre-defined formats [151], which is extremely challenging due to the hallucinations and unreliability of LLMs [152].

To tackle the above problems, we propose **GPiIoT**, a code generation system tailored for IoT application development by fine-tuning local small language models\* (SLMs) on IoT-specialized text-generation datasets. This approach has the following benefits: 1) The system overhead, privacy leakage, and network instability can be mitigated, as SLMs have smaller sizes and can be locally deployed without incurring heavy resource burdens. 2) SLMs tuned on IoT-specialized datasets can generate responses with significantly enhanced quality and higher relevance to the IoT domain [39]. 3) As our tuning datasets are well-structured text data, the tuned SLMs can produce intermediate outputs following the expected format with enhanced stability and avoid hallucinations.

We implement **GPiIoT**<sup>†</sup> with three tailored SLMs to handle different stages of IoT application development: TDSLML for Task Decomposition, RTSLM for Requirement Transformation, and CGSLM for Code Generation. Though tuning one SLM to handle all the tasks is possible, it is extremely challenging due to SLMs’ limited language understanding and processing capabilities [154, 155]. As shown in Fig. 4.1, TDSLML first decomposes an IoT application into multiple sub-tasks with detailed descriptions. Next, the descriptions are converted into well-structured specifications following pre-defined formats by RTSLM. Accordingly, CGSLM further generates a list of code snippets with detailed documentation. By sequentially executing the code based on the documentation, the IoT task can be solved. Note that we only

---

\*We consider SLMs as open-source language models that can be locally deployed and operated efficiently on commodity GPUs [153] (*e.g.*, Llama2-13b with INT8 quantization requires around 16 GB of GPU memory).

<sup>†</sup>The models and datasets are available at <https://github.com/lemingshen/GPiIoT>

fine-tune TDSLM and CGSLM as these two stages require IoT domain knowledge during inference while RTSML only needs basic language processing.

In practice, we face three significant technical challenges. 1) **Lack of high-quality data.** To the best of our knowledge, there are no IoT-oriented user-requirement-to-sub-task and sub-task-to-program text-generation datasets. Thus, we first construct two datasets containing IoT knowledge retrieved from various public sources, aiming to enhance the task decomposition and code generation abilities of TDSLM and CGSLM, respectively. Moreover, we design an IoT-oriented text data augmentation method to enhance the datasets’ quality and diversity, considering the unique characteristics (*e.g.*, sensor modality and resource heterogeneity) of IoT applications, thereby enhancing SLMs’ knowledge comprehension and code generation capabilities for IoT tasks. 2) **Domain misalignment between SLMs.** The decomposed tasks generated by TDSLM may fall beyond the scope that CGSLM can handle due to domain misalignment. This is because the two SLMs focus on different stages of IoT application development during tuning, which could lead to thematic inconsistencies in task interpretation and execution (§ 4.2.2). To tackle this, we propose a parameter-efficient co-tuning (PECT) paradigm featuring a multi-path Low-Rank Adaptation (LoRA) pipeline. Unlike conventional LoRA tuning that tunes adapters separately, our designed PECT paradigm enables collaborative fine-tuning of multiple SLMs with a shared base model but with different adapters, thereby mitigating the inconsistency issues and facilitating knowledge sharing between SLMs. 3) **Format incompatibility.** Decomposed tasks are typically described in natural language, while the expected inputs of CGSLM should be well-structured. If we directly use the decomposed task descriptions as prompts to generate code, CGSLM can not provide programs strictly following user requirements (§ 4.2.2). To address this, we leverage Chain-of-Thought (CoT) prompting [87] that instructs RTSML to transform the descriptions into well-structured specifications step by step. As such,

CGSLM can better handle the specifications to provide IoT-specialized solutions.

To evaluate **GPIoT**, we also propose *IoT Bench*, a benchmark to quantify LLMs’ capabilities in synthesizing IoT-related programs. Extensive experiments and a user study demonstrate that **GPIoT** can generate code adopting more IoT-specialized algorithms and outperform SOTA code LLMs in terms of task accuracy (more than 64.7% on average), memory usage (less than 310 MB on average), and user satisfaction. In summary, we make the following contributions:

- **GPIoT** presents the first code generation system tailored for IoT application development featuring privacy-preserving local SLMs tuned on IoT-specialized datasets.
- We create IoT domain text-generation datasets with a novel augmentation method tailored for the unique characteristics of IoT tasks, significantly enhancing the IoT knowledge comprehension ability of our tuned SLMs. We also construct *IoT Bench* to evaluate the capability of LLMs in synthesizing IoT-specialized programs.
- We propose the PECT paradigm, a new LLM tuning method that can collaboratively fine-tune multiple SLMs to mitigate their domain misalignment with facilitated knowledge exchange.

## 4.2 Background & Motivation

We first revisit existing code LLMs to underscore the importance of constructing tailored IoT-related LLMs. Then, we conduct some preliminary experiments on existing LLM+RAG methods to further motivate our work with several challenges we need to address.

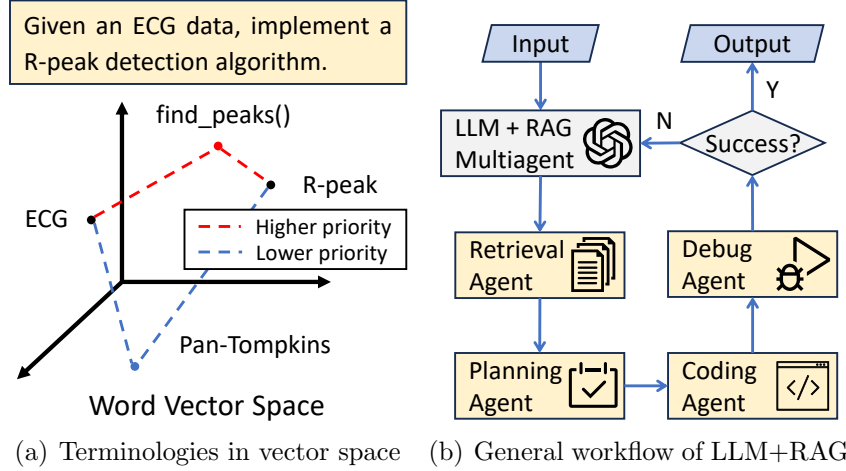


Figure 4.2: (a) Existing LLMs tend to prioritize general terms; (b) LLM+RAG systems require multiple cascaded agents.

#### 4.2.1 Code LLM and LLM+RAG

Existing code LLMs aim to synthesize programs and enhance software development efficiency and accuracy. While they perform well on general and simple programming tasks (*e.g.*, sorting algorithms), they often struggle with complex problems in the IoT domain. For example, when prompted to design an R-peak detection method for electrocardiogram (ECG) data, existing code LLMs can only use the `find_peaks()` function, which adopts a general peak detection algorithm rather than a dedicated one tailored for ECG data (*e.g.*, Pan-Tompkins [104]). The underlying reason is that IoT knowledge and programs only occupy a small proportion of the training dataset of code LLMs. As a result, despite being presented with abundant IoT terminologies in the prompt, LLMs still tend to prioritize and respond with more general words, due to their higher similarity (shorter distance in Fig. 4.2(a)) within the vector representation space.

LLM+RAG methods address this by retrieving domain knowledge for reference and establishing multiple cascaded agents to facilitate information transfer among modules. For example, in Fig. 4.2(b), multiple LLM-based agents are employed for

different tasks during development (*i.e.*, domain knowledge retrieval, task planning, coding, and debugging). Sophisticated prompt design and meticulously structured intermediate outputs are necessary to ensure that one agent’s output can be accurately parsed and interpreted by another agent. We conduct a preliminary experiment by prompting MapCoder [1], a multi-agent-based LLM+RAG framework, to synthesize programs for R-peak detection. We repeatedly generate 100 distinct versions of the programs and analyze them through code review and execution. Surprisingly, we find that only 28% of the programs adopt appropriate IoT-related algorithms to perform R-peak detection. This is because LLM+RAG requires sophisticated RAG design and user prompts. Otherwise, the retrieved knowledge is less accurate and relevant to the IoT context, and LLMs may fail to focus on the IoT domain and still provide simple and general solutions [150]. Moreover, this cascading process inevitably introduces noise and propagates errors [156], leading to a long self-recover time. More importantly, cloud LLMs suffer from bad network conditions, high costs, and privacy concerns.

To overcome these challenges, GPIoT fine-tunes SLMs on IoT-specialized text-generation datasets, as SLMs have smaller sizes and can be locally deployed. Additionally, by steering the parameter distribution towards the IoT domain via tuning, SLMs can focus on IoT-related semantic context, generating highly relevant responses that follow pre-defined formats with enhanced stability.

### 4.2.2 Preliminary Experiments & Findings

We conduct some preliminary experiments by separately fine-tuning two SLMs on our manually constructed datasets (§ 4.4.1), *i.e.*, the task decomposition dataset (TDD) and the code generation dataset (CGD). TDD aims to enhance TDSLM’s capability to break a *problem statement* proposed by the user into multiple *decomposed tasks* described in natural language. CGD aims to enhance CGSLM’s ability to generate

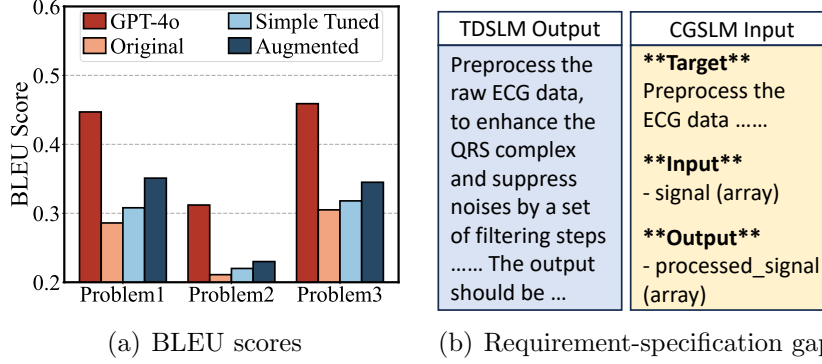


Figure 4.3: (a) Directly tuning SLM with simple augmentation only yields small improvements; (b) Gap between SLMs.

*code & documentation* for the user based on the *decomposed tasks*. However, we find it extremely challenging to ensure the correctness of the generated code. Note that we use Llama2-13b [118] as the default SLM for demonstration purpose.

**Lack of high-quality data.** Directly fine-tuning SLMs incurs little performance gain, even with data augmentation. We first deploy four models: GPT-4o, the original SLM, the SLM tuned on TDD, and the SLM tuned on augmented TDD via Evol-Instruct [157]. Then, we randomly select three IoT problems from TDD and input them into the four models to obtain a set of responses. Next, we measure the similarity between the generated responses and the human-crafted references as ground truth (*decomposed tasks*) using the BLEU score [158], where a larger value indicates higher semantic similarity. As shown in Fig. 4.3(a), GPT-4o achieves the highest score with an acceptable value for such a text-generation task. However, the scores achieved by tuned SLMs increase slightly even with augmented TDD. Further analysis reveals that the solutions provided by the SLMs are either irrelevant to the IoT domain or contain hallucinations. This is because traditional augmentation methods (*e.g.*, Evol-Instruct) focus on augmenting linguistic characteristics of the original text data, which may fall short of effectively capturing intricate relationships among IoT terminologies. *This motivates us to design an IoT-tailored text*

*augmentation method to enhance the quantity, quality, and diversity of the original dataset.*

**Domain misalignment.** Since TDSLM and CGSLM are tuned on distinct datasets for different tasks, domain misalignment occurs when used in tandem. Specifically, we feed the task descriptions generated by TDSLM into CGSLM to synthesize corresponding programs for each sub-task. Surprisingly, we find that only 53.4% of the programs can be successfully executed without bugs and only 10.6% of the programs adopt IoT-specialized algorithms for the IoT tasks. The main reason is that the two SLMs develop expertise in different domains with knowledge inconsistency during tuning, hindering the seamless integration of task decomposition and code generation. As a result, the responses generated by TDSLM may fall outside the scope that CGSLM can handle. *This motivates us to develop a knowledge-sharing strategy between the two SLMs during tuning so that they can reach a consensus when handling IoT tasks.*

**Format incompatibility.** TDSLM’s outputs (*decomposed tasks*) are described in natural language while CGSLM’s inputs (*task specifications*) should be well structured (Fig. 4.3(b)). When we directly feed TDSLM’s output into CGSLM, only 23.6% of the synthesized programs can be successfully executed. The rest exhibits higher uncertainty with a lack of confidence in mapping the input task specification to the desired code [159]. The reason is that CGSLM is more sensitive to well-formatted inputs as it has been tuned on our dataset with structured text. Though directly tuning TDSLM to generate well-structured task specifications can be a solution, we find it challenging due to the limited language processing capabilities of SLMs, which cannot be sufficiently enhanced through tuning alone. *This motivates us to develop a method to convert the task descriptions in natural language into well-organized specifications.*

To address the above challenges, we propose three key technical modules, *i.e.*,

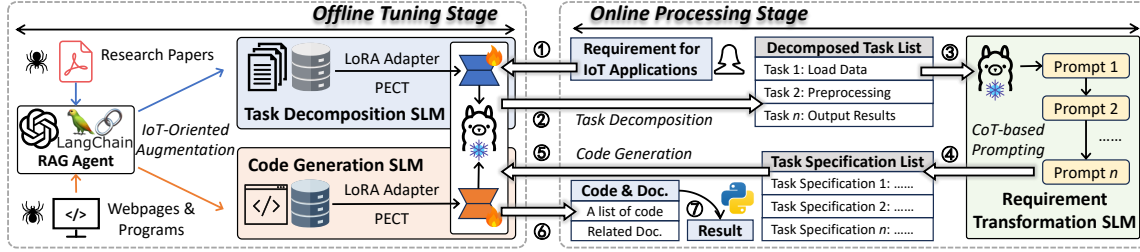


Figure 4.4: The system overview and workflow of GPIoT (All the local SLMs share the same foundation model).

an IoT-oriented text data augmentation method, a Parameter-Efficient Co-Tuning (PECT) paradigm with a multi-path LoRA pipeline, and a requirement transformation module.

### 4.3 System Overview

Fig. 4.4 illustrates the overall architecture of GPIoT, consisting of an offline tuning stage and an online processing stage.

**Offline Stage.** The offline tuning stage (the left part in Fig. 4.4) constructs two IoT-specialized datasets and fine-tunes TDSLM and CGSLM, which will be used for task decomposition and code generation in the online stage, respectively. We first build a RAG agent to extract knowledge and programs from various IoT-related public sources (*e.g.*, websites and articles) to **construct high-quality datasets**. Then, we augment the datasets by adopting our IoT-oriented augmentation method (§ 4.4.1) to enhance their quantity, quality, and diversity. Note that the RAG agent is only used for high-quality dataset construction during the offline stage. With the two augmented datasets, we fine-tune two SLMs via our PECT paradigm, where certain model parameters are collaboratively tuned through a multiple-path LoRA pipeline with two projection layers for task decomposition and code generation, respectively. Our PECT paradigm **mitigates the domain misalignment** between TDSLM and CGSLM with facilitated knowledge transfer and sharing.

**Online Stage.** The online stage (the right part in Fig. 4.4) aims to synthesize IoT-specific programs based on the user requirement for an IoT application development. Specifically, GPIoT first leverages *Task Decomposition SLM* (TDSLM) to decompose the IoT application into multiple manageable sub-tasks with detailed descriptions (①~②). Next, through CoT-based prompting techniques, the sub-task descriptions will be gradually **transformed into well-structured specifications** by *Requirement Transformation SLM* (RTSLM) (③~④). Next, for each sub-task, *Code Generation SLM* (CGSLM) accordingly generates a code snippet with documentation (⑤~⑥). Users can execute the code sequentially to realize the IoT application based on the instructions from the documentation (⑦).

**SLM Considerations.** We consider SLMs as open-source models that can be locally deployed and operated efficiently on commodity GPUs (*e.g.*, RTX 4070 Ti). This aligns with the practical constraints of normal users, where local models offer advantages in terms of cost, privacy, and independence from the cloud. Note that although there are three SLMs working simultaneously, *they share the same foundation model and differ only in some additional tunable parameters*, which only occupy 1% of all the parameters. Such a low-cost tuning and inference process stems from our PECT paradigm, avoiding significant overhead when deploying GPIoT on local devices.

## 4.4 System Design

### 4.4.1 Data Collection & Augmentation

Since there are no text-generation datasets in the IoT domain, we need to first construct a task decomposition dataset (TDD) and a code generation dataset (CGD). Note that our datasets contain Q&A pairs in textual form, fundamentally differing from conventional IoT datasets that typically contain pairs of sensor data and labels.

## Task Decomposition Dataset.

TDD contains pairs of “*problem statement*  $\rightarrow$  *decomposed tasks*”, aiming to enhance TDSLML’s task decomposition ability for IoT problems. The construction process consists of three stages: raw IoT-related text data collection, data formatting, and IoT-oriented text data augmentation.

**Raw Data Collection.** IoT-related research papers contain a huge quantity of high-quality SOTA applications and algorithms. Moreover, the systems proposed are comprehensive and functional, which can be decomposed into multiple modules with clear motivation and implementation details. Therefore, we download IoT-related papers from several public literature databases<sup>‡</sup> as our high-quality data sources, covering a wide range of IoT topics, such as communication, wireless sensing, edge computing, *etc.*

**Data Formatting.** We need to extract IoT knowledge from the papers and format it to pairs of “*problem statement*  $\rightarrow$  *decomposed tasks*”. Intuitively, we can regard the proposed system in each paper as an IoT problem, with its technical modules as the corresponding decomposed tasks. However, two challenges occur if we directly use such “*System*  $\rightarrow$  *technical modules*” pairs for tuning: 1) These module descriptions are typically lengthy, which exceed the context length of SLMs [160]. 2) These modules are still sophisticated, often containing multiple sub-systems. TDSLML may struggle to extract IoT-specialized technical concepts and accurately generate manageable components. To tackle this, our insight is that we regard each module as an individual problem, which can be further decomposed into several manageable sub-tasks. The disintegrated sub-tasks can be easily handled by TDSLML with reduced context length.

Fig. 4.5 shows the entire process of how we extract pairs of “*problem statement*

---

<sup>‡</sup>We download papers from public databases via our institution’s certification. The papers are for research only, adhering to ethical standards.

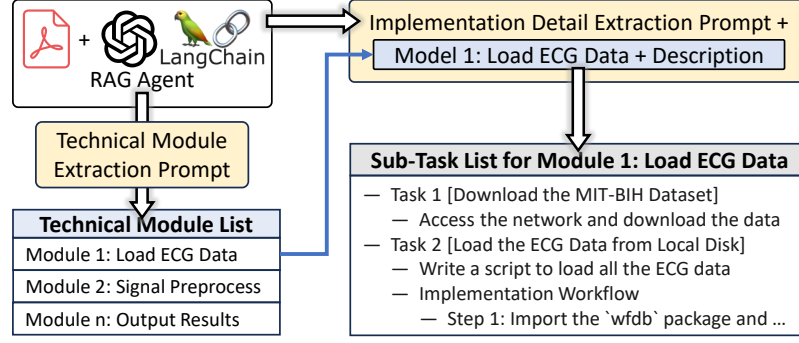


Figure 4.5: Task decomposition dataset construction.

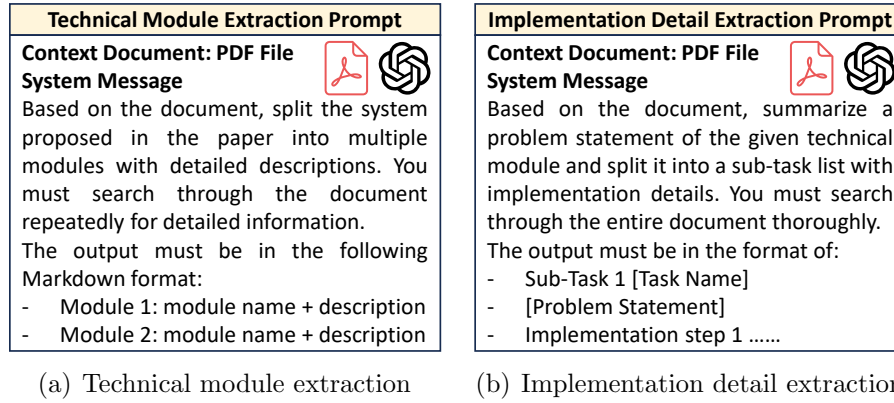


Figure 4.6: Prompts for paper information extraction.

→ *decomposed tasks*” from the papers. Specifically, we first build a RAG agent by combining the downloaded papers with an LLM (GPT-4o). Based on the provided context documents, we then prompt (Fig. 4.6(a)) the agent to split the proposed system in the paper into multiple technical modules with detailed descriptions. Next, for each technical module, we prompt (Fig. 4.6(b)) the agent to further decompose it into several sub-tasks with detailed implementation steps. As such, we encapsulate the *problem statement*  $p_i$  of each technical module and the corresponding sub-tasks  $t_i$  into a Q&A pair  $Q_i$  to construct a raw dataset  $\mathcal{D}_t$ :

$$\mathcal{D}_t = \{Q_1, Q_2, \dots, Q_{n_t}\}, \quad Q_i = (p_i, t_i) \quad (4.1)$$

where  $n_t$  is the total number of technical modules from all the papers. Fig. 4.8(a)

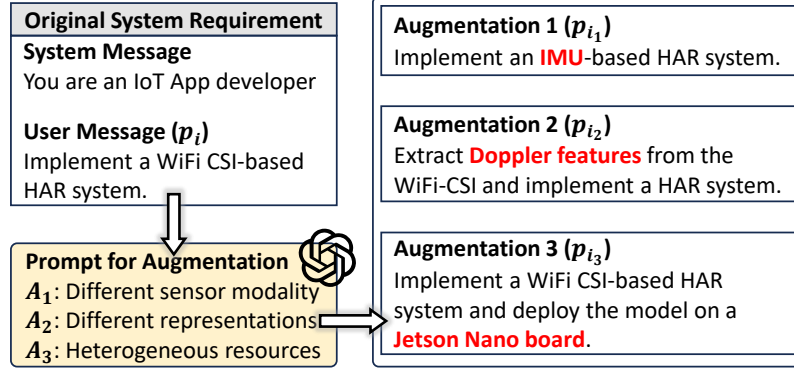


Figure 4.7: Examples of IoT-oriented data augmentation with different modalities, representations, and resource budgets

shows a data sample from the dataset. Note that each sub-task is separated by a blank line, allowing us to parse and split  $t_i$  into multiple task description strings for further code generation in a divide-and-conquer way.

**IoT-Oriented Data Augmentation.** As revealed in § 4.2.2, existing text augmentation methods are ineffective in the IoT domain as they focus on expanding language characteristics rather than IoT knowledge. As a result, IoT terminologies are still assigned lower priority during inference, preventing the tuned model from generating IoT-specialized solutions. To address this, we propose a novel IoT-oriented data augmentation method that considers unique properties of IoT applications, *i.e.*, sensor modality, data representation, and system resource heterogeneity, as shown in Fig. 4.7.

Our augmentation considers three aspects: 1) *Sensor modality*. For the same IoT problem, we can use different sensor modalities. For instance, to implement human activity recognition (HAR), we can utilize IMU data, WiFi CSI, *etc.* 2) *Data representation*. For the same modality, we can leverage distinct data representations to achieve the same task. For example, we can use WiFi CSI, 2D spectrograms, or extracted Doppler features to implement HAR. 3) *Resource heterogeneity*. For

| Task Decomposition Data Example                                                                                                                                                                                                                                                                                                                                                                                           | Sensor Data Modality Augmentation Prompt                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>System Message</b><br>You are an IoT application developer. Decompose the problem statement into a task list with implementation details. The output must be in the format of:<br>- Sub-Task 1 [Task Name]<br>- Implementation step 1 .....<br><br><b>User Message</b> ( $p_{i_j}$ )<br><Problem Statement><br><br><b>Assistant Message</b> ( $t_{i_j}$ )<br>- Sub-Task 1 [Task Name]<br>- Implementation step 1 ..... | <b>System Message</b><br>You are a professional prompt writer. Rewrite the prompt into a diverse version by replacing the sensor data modality with a different one (e.g., WIFI --> IMU). Direct output three different versions of the rewritten prompt separated by a blank line in the following format:<br><rewritten prompt1><br><rewritten prompt2><br><br><b>User Message</b> ( $p_{i_j}$ )<br><Problem Statement> |

(a) Task decomposition data sample      (b) Augment sensor modality

Figure 4.8: (a) Tuning data sample for task decomposition. (b) Prompt for sensor modality augmentation.

the same task, various IoT devices require heterogeneous system resources. When deploying an AI model, smartphones typically have less memory than PCs, requiring model optimization methods. Based on the three aspects, we prompt (Fig. 4.8(b)) GPT-4o to rewrite and augment each *problem statement* from  $\mathcal{D}_t$ . To generate reference decomposed tasks, we build a search agent to retrieve relevant IoT domain knowledge and prompt it to produce results. We then manually filter out incorrect results and craft the formats (*i.e.*, each sub-task is separated by a blank line as aforementioned). The augmented dataset  $\mathcal{D}'_t$  is:

$$\mathcal{D}'_t = \bigcup_j^3 \{(p_{i_j}, t_{i_j}) \mid p_{i_j} = A_j(p_i), t_{i_j} = G(p_{i_j})\}, \forall p_i \in \mathcal{D}_t \quad (4.2)$$

where  $A_j(\cdot)$  is the  $j$ -th type of augmentation operation and  $G(\cdot)$  is the black-box function of GPT-4o.

**Remark.** We take the diversity of both language expression and IoT characteristics into account, demonstrating significant performance improvement in task decomposing (§ 4.6.5). Note that the data collection and augmentation processes are both performed offline.

| API Reference - A Function from SciPy                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Example Gallery<br>A Usage Sample                                                                                                                                                                                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>find_peaks(x, height=None, threshold=None)</pre> <p>This function takes a 1-D array and finds all local maxima by simple comparison of neighboring values.</p> <p><b>Parameters</b></p> <ul style="list-style-type: none"> <li>- x (array): A signal with peaks.</li> <li>- height (int): required height of peaks.</li> <li>- threshold (int): required threshold of peaks</li> </ul> <p><b>Returns</b></p> <ul style="list-style-type: none"> <li>- peaks (array): indices of peaks in x.</li> </ul> <p><b>Examples</b></p> <pre>&gt;&gt;&gt; import numpy as np &gt;&gt;&gt; from scipy.signal import find_peaks &gt;&gt;&gt; x = np.random.rand(2000) &gt;&gt;&gt; peaks, _ = find_peaks(x, height=0)</pre> | <p><b>Universal Non-Uniform Random Number Sampling</b></p> <p><b>Introduction</b></p> <p>Random variate generation deals with algorithms to generate random variates from various distributions .....</p> <p><b>Workflow</b></p> <p>First ....., Then ....., Finally .....</p> <p><b>A Simple Example</b></p> <pre>&gt;&gt;&gt; import scipy &gt;&gt;&gt; x = ..... &gt;&gt;&gt; output = .....</pre> |

Figure 4.9: An example of a Python package’s website.

### Code Generation Dataset.

CGD contains pairs of “*task specification*  $\rightarrow$  *code & documentation*”, aiming to enhance CGSLM’s ability in generating IoT-related code for decomposed tasks. The data construction includes two stages: raw data collection and target diversity-aware augmentation for different code generation tasks.

**Raw Data Collection.** Open-source IoT-related Python<sup>§</sup> packages (*e.g.*, SciPy [161]) contain abundant hand-crafted IoT algorithms and applications with high performance, which can serve as our data sources. Thus, we first collect numerous public repositories from GitHub and extract Python packages they used, covering areas of signal processing, machine learning, and data processing (IoT data I/O and visualization). We then build a web crawler to automatically retrieve information from each package’s official website.

A package’s website (Fig. 4.9) typically contains two parts: 1) *API reference* includes a list of modules (*i.e.*, functions and classes) with comprehensive guidance on how to use them effectively in code. For example, `find_peaks()` is a function from the SciPy package, which identifies the local maxima (peaks) in an input signal

<sup>§</sup>We focus on Python since it is a cross-platform programming language.

array and returns the indices of the peaks. We denote the detailed information of each module as its metadata,  $m_i$ . 2) *Example gallery* provides practical usage samples of how to use various modules and features of the package to implement specific algorithms. For instance, a usage sample provides detailed documentation with code about performing universal non-uniform random number sampling using the SciPy package in an end-to-end manner. We denote the detailed information of each usage sample as its metadata,  $u_j$ . By combining these two types of metadata, we form a raw dataset  $\mathcal{D}_c$ :

$$\mathcal{D}_c = \{m_i \mid \forall i \in \{1, 2, \dots, n_m\}\} \cup \{u_j \mid \forall j \in \{1, 2, \dots, n_u\}\} \quad (4.3)$$

where  $n_m$  and  $n_u$  is the total number of modules and usage samples.

**Target Diversity-Aware Augmentation.** This augmentation aims to enhance the diversity of the metadata in  $\mathcal{D}_c$ . We target each module in the packages for two text-generation tasks: 1) *Module Description*: providing detailed descriptions of a module and 2) *Module Implementation*: writing code & documentation to demonstrate usage samples of the module. Since the example gallery already contains abundant algorithms with sample code and detailed descriptions, they can be directly used as code generation tasks.

1) *Module Description*. We format the task specification to “Provide detailed descriptions of <module> from <package>.” The corresponding reply contains the module metadata in a pre-defined format. This Q&A mapping relation from a *task specification*  $t_i$  to a *module description*  $m_i$  is expressed as:

$$\mathcal{D}_1 = \{t_i \rightarrow m_i\} \quad (4.4)$$

Such a “*task specification*  $\rightarrow$  *module description*” mapping can teach CGSLM to be familiar with the module’s information, strengthening the semantic correlation between the module’s name and the detailed descriptions. Fig. 4.10(a) illustrates a data sample from  $\mathcal{D}_1$ .

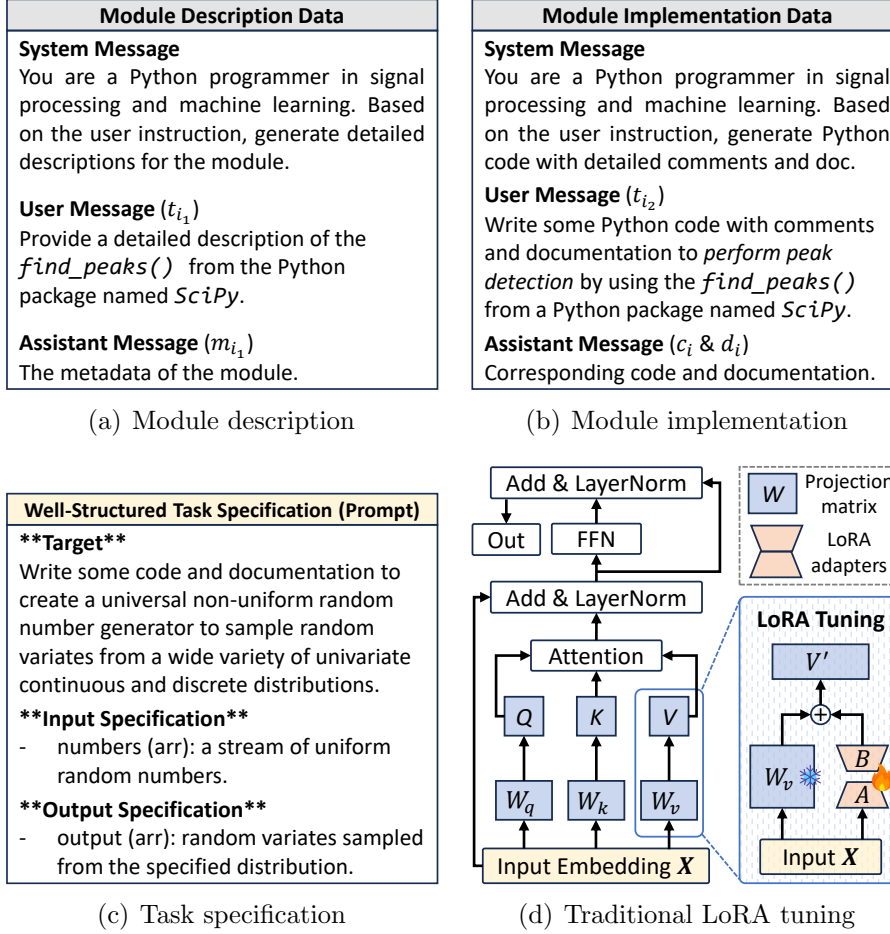


Figure 4.10: (a) & (b) Two tuning data samples. (c) Well-structured task specification. d) Traditional LoRA tuning paradigm.

2) *Module Implementation*. We format the task specification to “Write some Python code with comments and documentation to perform `<target>` by using `<module>` from `<package>`.” The corresponding reply contains the sample code and documentation that provides the workflow and guidance on how to execute the code. To obtain well-structured documentation, we prompt GPT-4o to format the module’s metadata into Markdown. This Q&A mapping relation from a *task specification*  $t_i$  to the corresponding *module implementation* (i.e., code  $c_i$  and documentation  $d_i$ ) is:

$$\mathcal{D}_2 = \{t_i \rightarrow (c_i, d_i) \mid d_i = G(m_i)\} \quad (4.5)$$

Such a “*task specification*  $\rightarrow$  *module implementation*” mapping relationship aims to enhance CGSLM’s capability in generating code and documentation according to the module specification. Fig. 4.10(b) shows a data sample from  $\mathcal{D}_2$ .

3) *Example Implementation.* We format the task specification to a well-structured Markdown format as shown in Fig. 4.10(c), including the task target and the I/O specifications for the expected code. Correspondingly, we prompt GPT-4o to convert the usage sample’s metadata into well-structured documentation. This Q&A mapping relation from a *task specification*  $t_j$  to the code  $c_j$  and documentation  $d_j$  can be expressed as:

$$\mathcal{D}_3 = \{t_j \rightarrow (c_j, d_j) \mid d_j = G(u_j)\} \quad (4.6)$$

This aims to enhance CGSLM’s ability in generating IoT-related code and detailed documentation following well-structured task specifications. Ultimately, by concatenating all three augmented datasets, the final CGD  $\mathcal{D}'_c$  becomes:

$$\mathcal{D}'_c = \mathcal{D}_1\{t_i \rightarrow m_{i_1}\} \cup \mathcal{D}_2\{t_i \rightarrow (c_i, d_i)\} \cup \mathcal{D}_3\{t_j \rightarrow (c_j, d_j)\} \quad (4.7)$$

### ***IoT Bench***

To evaluate LLMs’ abilities in task decomposition and code generation for IoT applications, we create *IoT Bench*, a benchmark of text-generation tasks in the IoT domain. Specifically, we choose 100 samples from TDD and CGD with manually created test cases, covering various IoT topics (*e.g.*, signal processing, edge AI, *etc.*). All the selected data samples are first manually filtered to ensure correctness and relevance to the IoT domain. Then, we format the sub-tasks separated by a blank line in between. Note that although many SOTA benchmarks (*e.g.*, HumanEval [48]) can also evaluate LLMs’ code generation abilities, they are not tailored to IoT tasks. Besides, the data in *IoT Bench* is excluded from the tuning processes (§ 4.4.2) to test the generalizability of the tuned SLMs.

#### 4.4.2 Parameter-Efficient Co-Tuning (PECT)

With the two augmented datasets ( $\mathcal{D}'_t$  and  $\mathcal{D}'_c$ ), our next step is to fine-tune TDSLM and CGSLM to enhance their ability in task decomposition and code generation, respectively. In the following, we first introduce the traditional LoRA tuning method [60] for SLMs, and then explain our Parameter-Efficient Co-Tuning paradigm.

##### LoRA Tuning.

Fig. 4.10(d) shows the Low-Rank Adaptation (LoRA) tuning process of a Transformer block [162]. Specifically, each Transformer block in an LLM contains two main components: a self-attention mechanism and a feed-forward network (FFN), both of which are followed by residual connections and layer normalization. The self-attention features three tunable weight matrices ( $W_q$ ,  $W_k$ , and  $W_v$ ) to capture contextual relationships between input embeddings, while the FFN processes the outputs from the attention mechanism to refine the feature representations. In conventional LLM full-tuning, the entire weight matrices are updated, leading to extensive GPU memory requirements and high computational costs. Instead of fully updating the weight matrices, LoRA reduces the number of tunable parameters [163, 164], where two low-rank matrices  $\mathbf{A}$  and  $\mathbf{B}$  (*i.e.*, LoRA adapters) are inserted alongside the weight matrix. Given an input  $\mathbf{X}$ , the tuning process can be expressed as:

$$\mathbf{V}' = (\mathbf{W}_v + \mathbf{B}\mathbf{A}) \cdot \mathbf{X} \quad (4.8)$$

This reduces the computational burden by only updating the smaller low-rank matrices  $\mathbf{A}$  and  $\mathbf{B}$ , significantly cutting down resources.

However, as demonstrated in § 4.2.2, domain misalignment arises when separately tuning TDSLM and CGSLM using this vanilla LoRa method. This is because they focus on two distinct text-generation tasks with different semantic attention, thereby hindering GPIoT from synthesizing IoT-related programs. To tackle this

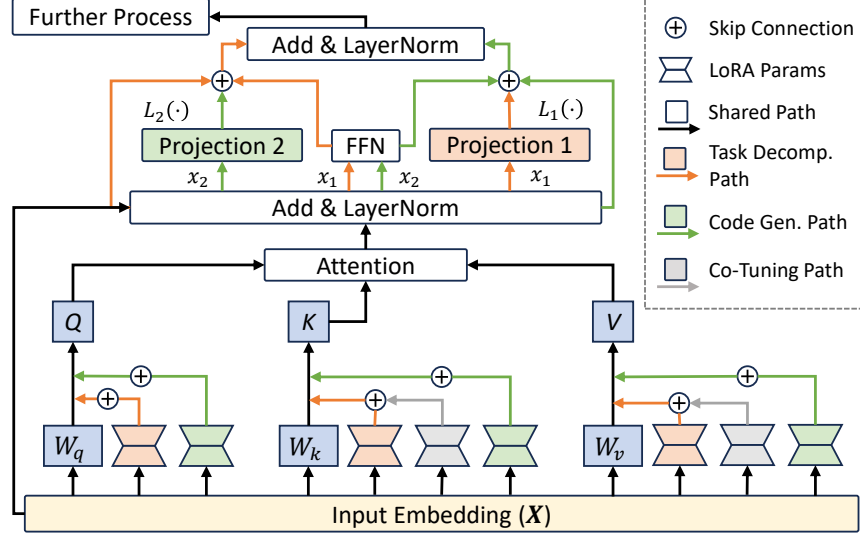


Figure 4.11: PECT in one Transformer block with both independently and collaboratively tuned LoRA adapters.

issue, we propose a parameter-efficient co-tuning (PECT) paradigm. *Unlike conventional LoRA tuning that tunes adapters separately, PECT enables collaborative fine-tuning of several SLMs with a shared base model but with different LoRA adapters.* PECT features a Multi-Path LoRA Pipeline (MPLP) and two lightweight projection layers, which can promote information sharing between TDSLM and CGSLM, thereby narrowing the semantic comprehension gap between task decomposition and code generation.

### Multi-Path LoRA Pipeline.

MPLP selects a subset of shared LoRA adapters to be collaboratively tuned by TDSLM and CGSLM, with another adapter set independently tuned.

**Pipeline Construction.** In the lower part of Fig. 4.11, we create three pipelines of LoRA adapters in each Transformer block. Two pipelines (the orange one and the green one) are independently tuned by TDSLM and CGSLM with respect to TDD and CGD. The other pipeline (the gray one) is co-tuned on both TDD and CGD. For example, given a data sample from TDD, only the orange LoRA adapters and

gray LoRA adapters are updated, as shown in Fig. 4.11. Note that we only assign the shared adapters beside the key and value weight matrices ( $\mathbf{W}_k, \mathbf{W}_v$ ). The insight behind this is that the value vector provides the information to be activated based on the key vector [162]. In other words, the mapping from *problem statement* to *decomposed tasks* and the mapping from *task specification* to *code & documentation* are determined by the key and value vectors in TDSLML and CGSLM, respectively. Domain misalignment is thus caused by such different mapping relations during tuning on distinct datasets with disparate semantic focuses. Therefore, by co-tuning the shared adapters integrated into the key and value vectors, the mapping relations will be shared between the two SLMs, allowing TDSLML’s outputs to align with CGSLM’s scope.

**Co-Tuning.** In Fig. 4.11, we designate the orange line as the task decomposition path (TDP), through which only data from TDD will pass. The green line is the code generation path (CGP), through which only data from CGD will pass. The grey line represents the co-tuning path, through which all data will pass. During co-tuning, the LoRA adapters will be tuned either independently or collaboratively, depending on the path they occupy. Specifically, take the adapter alongside the projection matrix  $\mathbf{W}_k$  as an example, the key vectors after projection in the two paths are calculated by:

$$\begin{aligned} \mathbf{K}_1 &= (\mathbf{W}_k + \mathbf{B}_1\mathbf{A}_1 + \lambda \cdot \mathbf{B}_c\mathbf{A}_c) \cdot \mathbf{X} \\ \mathbf{K}_2 &= (\mathbf{W}_k + \mathbf{B}_2\mathbf{A}_2 + (1 - \lambda) \cdot \mathbf{B}_c\mathbf{A}_c) \cdot \mathbf{X} \end{aligned} \quad (4.9)$$

where  $\mathbf{X}$  is the input text embedding,  $\mathbf{K}_1$  and  $\mathbf{K}_2$  are the key vectors within TDP and CGP, respectively.  $\mathbf{B}_1\mathbf{A}_1$  and  $\mathbf{B}_2\mathbf{A}_2$  are the parameters of LoRA adapters independently tuned within the two paths, respectively.  $\mathbf{B}_c\mathbf{A}_c$  are the LoRA adapters collaboratively tuned by the two paths.  $\lambda$  is a hyper-parameter to balance the data flow between the two paths. During the co-tuning process, we first randomly sample data from TDD and CGD. Next, if the data is sampled from TDD, it will pass

through TDP; otherwise, it will pass through CGP. We then calculate the loss and update the corresponding LoRA adapters based on the source of the data sample.

**Remarks.** By orchestrating the independent and collaborative tuning paths, MPLP dismantles the information barrier between TDSLM and CGSLM, fostering their consensus during inference and thereby alleviating the misalignment issue.

### Projection Layers.

To further enhance knowledge sharing between the two SLMs during tuning, we create two projection layers for the two paths. We place the projection layers in parallel with the FFN layers in the Transformer block. As such, they can serve as extra FFNs that apply non-linear transformations to the attention representations, thereby enhancing token-level feature extraction and increasing the complexity of the model’s learning capabilities. By receiving representations from the other path with enhanced non-linearity, the cross-domain IoT knowledge comprehension capabilities of the two SLMs will be further strengthened. Specifically, take TDP as an example, as shown in the upper part of Fig. 4.11, with the obtained value (denoted as  $x_1$ ) from the LayerNorm in TDP, we feed it into a projection layer  $L_1(\cdot)$ . The output is then added with the FFN’s output  $F(x_2)$  and  $x_2$  in CGP. The sum will be sent to the next Transformer block for further processing. Such a knowledge transfer process can be expressed as:

$$\begin{aligned} x'_1 &= x_1 + F(x_1) + \gamma \cdot L_2(x_2) \\ x'_2 &= x_2 + F(x_2) + (1 - \gamma) \cdot L_1(x_1) \end{aligned} \quad (4.10)$$

where  $x'_1$  and  $x'_2$  are the final output of the two paths.  $x_1$  and  $x_2$  are the input attention representations from TDP and CGP, respectively.  $F(\cdot)$  represents the FFN layer,  $L_1(\cdot)$  and  $L_2(\cdot)$  are the projection layers in the two paths, and  $\gamma$  is a hyper-parameter to balance the knowledge-sharing between the two paths. Note that each projection layer has the same architecture as the FFN, consisting of two

fully connected layers and a non-linear SwiGLU function [165].

**Remarks.** By combining independent and collaborative tuning of LoRA adapters with the projection layers, PECT optimizes the task-specific performance of TDSLML and CGSLM while minimizing domain conflicts. As a result, the decomposed tasks generated by TDSLML will have closer semantic alignment with CGSLM and thus can be better handled. Note that during both tuning and inference, the SLMs share the same foundation model architecture, with the only difference being the LoRA parameters shown in Fig. 4.11. Therefore, our proposed PECT paradigm fundamentally differs from those traditional PEFT approaches that tune SLMs separately.

### 4.4.3 Requirement Transformation

When cascading TDSLML and CGSLM together, a huge gap exists between TDSLML’s outputs (*decomposed task descriptions*) and CGSLM’s inputs (*task specifications*). The task descriptions are typically natural language while the task specifications are well-structured. Directly feeding the task descriptions into CGSLM will lead to sub-optimal performance of the generated code. To fill this gap, we leverage RTSLM to transform the descriptions into well-structured specifications. Considering that RTSLM has limited IoT knowledge comprehension ability, we enhance it with RAG and several CoT-based prompts to perform requirement transformation.

**RAG Construction.** To enhance RTSLM’s ability in understanding IoT domain knowledge during requirement transformation, we first transform all the downloaded papers into a text embedding database. Then, armed with such an IoT knowledge database, we build a RAG agent based on RTSLM to retrieve relevant context for reference. As such, RTSLM can better comprehend and handle IoT terminologies in the task descriptions during requirement transformation.

**CoT Prompting.** Fig. 4.10(c) shows an example of a well-structured task specification for code generation, consisting of three parts: task target, input and out-

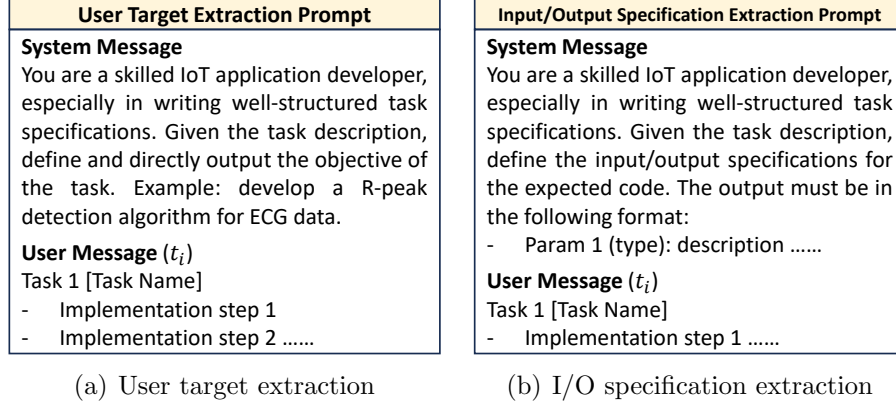


Figure 4.12: CoT-based prompts for RTSLM.

put specifications of the expected code. For each decomposed task  $t_i$  generated by TDSLML, we prompt the agent to generate such well-structured specifications step-by-step. Specifically, we first prompt (Fig. 4.12(a)) the agent to summarize a target for the task. Next, we further instruct (Fig. 4.12(b)) the agent to generate a list of parameter descriptions for the input and output of the expected code. Each single parameter description item contains the parameter name, the parameter type, and a brief explanation of its meaning. For example, “signal (numpy.ndarray): the raw ECG data collected from patients with noises.” Finally, RTSLM reorganizes and formats the above information into a well-structured task specification, which will be further handled by CGSLM to generate corresponding code snippets and documentation.

**Remarks.** Note that tuning is excluded in this process since it only needs basic language comprehension and processing capabilities of RTSLM. Therefore, RTSLM shares the same base model without additional tunable LoRA parameters to perform the transformation.

## 4.5 Experiment Setup

### 4.5.1 Implementation

**System Configurations.** We deploy GPIoT on an edge PC equipped with an RTX 4090 GPU (24 GB). We use selenium [166] to create a web crawler for data retrieving from public websites. To perform data formatting and augmentation, we construct an agent based on GPT-4o and LangChain [91]. For SLM tuning, we use a high-performance cloud server with an NVIDIA A100 GPU (80 GB).

**Hyper-parameters.** TDD contains 36,098 pairs of “*problem statement*  $\rightarrow$  *decomposed tasks*”. CGD contains 35,419 pairs of “*task specification*  $\rightarrow$  *code* & *documentation*”. Llama2-13b [118] with INT8 quantization serves as the foundation model and is fine-tuned via LoRA [60], with a rank of 64 and a dropout rate of 0.1. The number of tuning epochs is 5, with an initial learning rate of 0.0001, varied by a cosine learning rate scheduler. The  $\lambda$  in Eq. 4.9 and the  $\gamma$  in Eq. 4.10 are both set to 0.5 by default. The tuning process takes around 80 GPU hours. Since TDSLM, RTSLM, and CGSLM share the same foundation model, only about 16 GB of GPU memory is needed for the whole system, which is affordable for a commodity GPU [167].

### 4.5.2 IoT Applications

Considering the different technologies required during development, we select three IoT applications, focusing on healthcare and edge computing. 1) **Heartbeat Detection (HD)** is essential for continuously monitoring patient vitals with enhanced healthcare and ensuring timely intervention in case of abnormalities [68]. We instruct GPIoT to develop a heartbeat (R-peak) detection algorithm and test it on the MIT-BIH dataset [100]. 2) **Human Activity Recognition (HAR)** [73, 168, 169, 170, 171] deployed on edge devices is important for real-time analysis

of daily human activities. We instruct **GPIoT** to develop a WiFi-based HAR model using the WiAR dataset [172] and deploy it on a Jetson Nano board that has limited resources [173]. 3) **Multimodal HAR** leverages different sensors to capture complementary information, thereby enhancing HAR systems’ robustness and versatility [174]. We instruct **GPIoT** to construct a multimodal HAR model based on the Harmony dataset [69], which contains three sensor modalities: audio [175], depth camera, and radar [18].

**Notes:** HD requires signal processing methods, HAR demands technologies in both signal processing and machine learning, and multimodal HAR necessitates advanced multimodal processing algorithms. Though we use HD as an example for demonstration throughout the paper, all the tasks are unseen to **GPIoT**.

## 4.6 Evaluation

### 4.6.1 Metrics

We compare the programs synthesized by **GPIoT** and several baselines by measuring the following evaluation metrics.

**HD.** 1) *Precision*: The fraction of correctly detected R-peaks out of all detected peaks:  $\frac{TP}{TP+FP}$ . 2) *Recall rate*: The proportion of correctly detected R-peaks out of all actual R-peaks:  $\frac{TP}{TP+FN}$ . The larger these two metrics are, the more accurate the heartbeat detection becomes.

**HAR.** 1) *Classification accuracy*: The portion of the test data that is correctly classified based on the label. A higher accuracy implies a more robust and accurate HAR model. 2) *GPU memory usage*: The amount of GPU memory used during model inference. 3) *Inference time*: The time it takes from feeding the data into the code to the generation of the recognition result. The less memory and inference time consumed, the more resource-efficient the HAR model is.

### 4.6.2 Baselines

Given the same user problem, we input it into the following baselines to compare their performance with **GPIoT (GT)**. 1) **GPT-4o (G4)** [38] is an advanced LLM from OpenAI, optimized for instruction following and code generation tasks. 2) **DeepSeek-Coder (DC)** [134] is a high-performance code LLM, particularly effective in understanding and generating programming code across various domains. 3) **CodeLlama-34b (CL)** [49] is a specialized version of the Llama designed to generate, understand, and assist with coding. 4) **WizardCoder-33b (WC)** [85] incorporates complex instruction fine-tuning by adopting evolving instructions. 5) **CodeQwen-7b (CQ)** [176] is the Code-Specific version of Qwen1.5, which is a decoder-only LLM pre-trained on a large amount of data of programs. 6) **GitHub Copilot (GC)** [41] is an AI-powered code generation tool that assists developers by suggesting code snippets and functions. 7) **MapCoder (MC)** [1] is an LLM+RAG-based code generation framework that cascades multiple LLM-based agents to solve competitive problems, where GPT-4o is selected as the built-in LLM.

We access GPT-4o and DeepSeek-Coder via API keys, interact with GitHub Copilot via Visual Studio Code’s chat window, and deploy the rest on an edge server. Note that to our best, currently there is no LLM-based program synthesis system tailored for IoT application development. Therefore, we choose some SOTA code generation LLMs and systems as our baselines.

### 4.6.3 Application Evaluation

With the two problem statements (Fig. 4.13) designed for the three IoT applications, we input them into **GPIoT** and the baselines to synthesize 20 different programs for each task. We then evaluate their performance based on the metrics described in § 4.6.1.

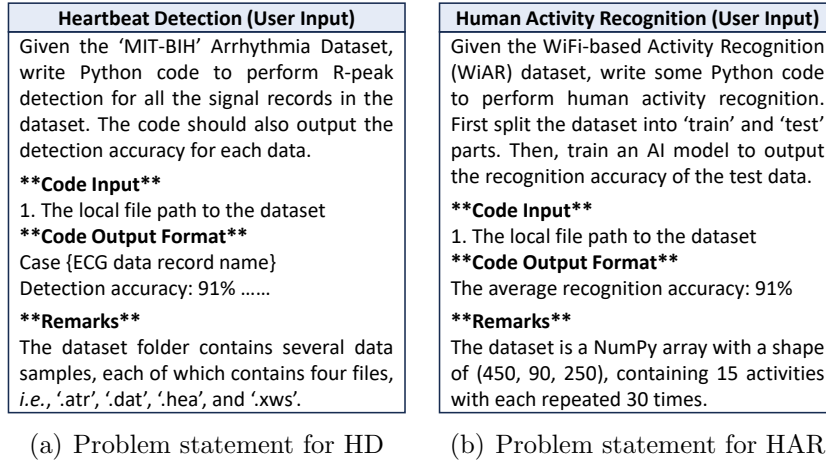


Figure 4.13: Problem statements of the two applications (HAR and multimodal HAR share a similar prompt).

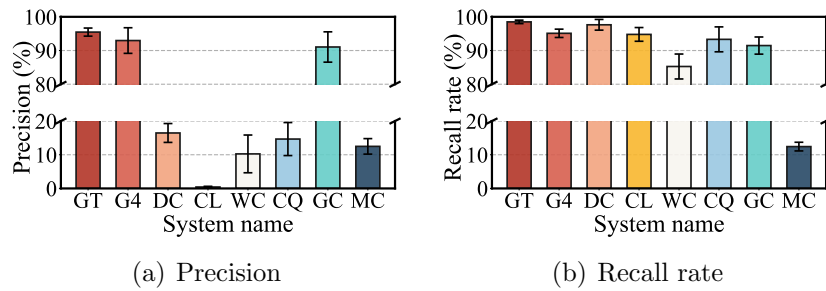


Figure 4.14: The overall performance of HD.

HD

As shown in Fig. 4.14, the code generated by GPIoT significantly outperforms all the baselines, with an average precision gain of 64.7% and an average RR increase of 16.9%. It’s also worth noting that CodeLlama, WizardCoder, and CodeQwen achieve moderate RR (above 80%) but exhibit lower precision. With further analysis of the code, we find that they all adopt a simple peak detection function, `scipy.signal.find_peaks()`, which typically fails when handling abnormal ECG data from patients. As a result, the detection results contain numerous false positives with low precision. Additionally, after reviewing the code generated by MapCoder,

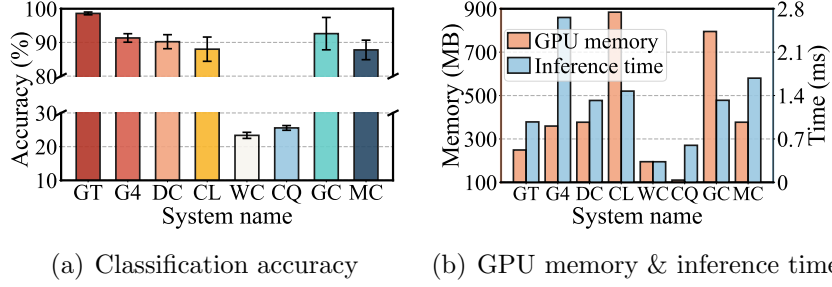


Figure 4.15: The overall performance of HAR.

we observe that it incorporates more advanced heartbeat signal processing algorithms (*e.g.*, bandpass filtering and adaptative thresholding). However, the final program exhibits a significant performance drop compared with other baselines. This is because heartbeat detection is a relatively simple IoT application that does not require highly sophisticated planning and iterative debugging. Integrating many advanced algorithms into a simple signal-processing program may lead to inconsistency issues. In other words, the heartbeat signal may be over-processed by these algorithms, leading to degraded performance. [177, 178]. On the contrary, the code generated by GPIoT utilizes dedicated algorithms (*e.g.*, Pan-Tompkins) for R-peak detection due to embedded IoT domain knowledge during tuning, consistently achieving high precision and RR.

## HAR

In this evaluation, for all the generated HAR models, we set the training epochs to 10 and the batch size to 32 for a fair comparison. Besides, during our implementation, we find that after around 15 training epochs, all the models gradually converge. Therefore, we compare the model performance at the 10th epoch. As shown in Fig. 4.15(a), the program synthesized by GPIoT achieves a 17.2% higher accuracy with 47.8% less GPU memory and 38.3% shorter inference time on average. By analyzing the generated code, we find GPIoT applies: 1) a data preprocessing method,

Butterworth low-pass filtering, on the WiFi data, considering that the low-frequency components of WiFi CSI are primarily influenced by human activities [179]. 2) an augmentation method tailored for IoT data (*e.g.*, time-frequency masking [180]) on the WiFi signal to further enhance the diversity of the dataset. 3) a CUDA optimization mechanism [181] to reduce GPU memory usage while enhancing runtime efficiency. In contrast, the baselines directly input raw WiFi data into HAR models without GPU optimization, leading to diminished performance and heightened memory consumption. Moreover, the error bar of **GPIoT** is smaller than that of the baselines, indicating that **GPIoT** generates more stable responses with a more robust performance of the synthesized program. Note that MC can synthesize programs with competitive performance since HAR is a more complex application. Such advancements of **GPIoT** originate from our meticulously crafted IoT-specialized datasets and the PECT paradigm, which embeds abundant IoT domain knowledge from our datasets into the tuned SLMs for more consistent outputs.

Additionally, considering that our HAR model is deployed on edge devices with various resource constraints [182, 183, 27, 184, 128], we involve different resource requirements in the prompts for the generated code to facilitate a more comprehensive evaluation. Specifically, we first design a base prompt: “I need to deploy the HAR model on Jetson Nano” (**P1**) without any resource specifications. Based on **P1**, we then create three variations by adding different resource constraints: “Do not consider any resource constraints but only model accuracy” (**P2**), “The GPU memory usage should not exceed 200 MB” (**P3**), and “The GPU memory should not exceed 50 MB” (**P4**). We then instruct **GPIoT** to synthesize 20 different versions of programs using each of the prompts above. After executing the programs with the same configurations, we record the average classification accuracy and GPU memory usage, as shown in Fig 4.19(b). We find that: 1) Given P2, **GPIoT** can construct an AI model with a large number of parameters to achieve ultimate performance, with

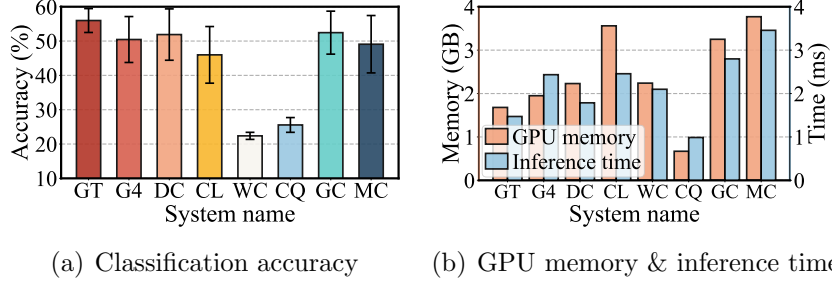


Figure 4.16: The overall performance of multimodal HAR

its classification accuracy approaching nearly 100%. 2) Given P3, **GPIoT** can adopt a smaller model within the resource budget (*i.e.*, GPU memory usage not exceeding 200 MB) with a slight performance drop. 3) Given P4, **GPIoT** employs a highly optimized model requiring only approximately 50 MB of GPU memory, resulting in an acceptable accuracy drop of 5%. These results demonstrate that **GPIoT** can generate code tailored to different resource budgets, stemming from our *IoT-Oriented Data Augmentation* method, which augments data samples considering resource heterogeneity of target devices in IoT applications.

## Multimodal HAR

We further instruct **GPIoT** to synthesize programs for the multimodal HAR application, aiming to evaluate its programming ability for more complex tasks. As shown in Fig. 4.16, compared with the baselines, the program synthesized by **GPIoT** achieves an average accuracy improvement of 13.44% while requiring moderate GPU memory and inference time. After reviewing the source code, we find that both **GPIoT** and the baselines train three encoders to first extract useful features from different modalities, followed by a classifier to recognize the corresponding activity. However, the program synthesized by **GPIoT** adopts some model optimization methods (*e.g.*, quantization or pruning) and data augmentation methods tailored for IoT sensor data (*e.g.*, time-frequency masking). As such, the synthesized program can train

a memory-optimized model while maintaining high classification accuracy. These results indicate that, benefiting from our SLM tuning, the program synthesized by GPIoT can incorporate more IoT-specific data processing and model optimization algorithms, thereby achieving high performance even for multimodal HAR.

#### 4.6.4 Breakdown Evaluation

We separately evaluate TDSLM and CGSLM on *IoT Bench* to explore the effectiveness of fine-tuning in the IoT domain.

##### Metrics

We adopt different metrics for the two SLMs.

**TDSLM.** 1) *BLEU*: we measure the BLEU score [158] between the generated decomposed tasks and the reference from *IoT Bench*. A larger BLEU score indicates higher semantic similarity and higher task decomposition quality. 2) *Format Correctness Rate* (FCR): the portion of TDSLM’s outputs that correctly separate each decomposed task with a blank line for the convenience of further processing. This aims to quantify TDSLM’s instruction-following and text-formatting abilities. 3) *Sub-Task Completeness* (STC): we invite 10 IoT experts to assess the extent to which the decomposed tasks cover all essential parts of the application based on the reference.

**CGSLM.** 1) *Code embedding similarity*: we use CodeT5+ [185] to convert code snippets into embeddings and compute the cosine similarity between embeddings of the generated and reference code. A higher similarity indicates a stronger ability to generate IoT-related code. 2) *Pass@k*: we measure the pass@k value by calculating the portion of programs that pass all the test cases. A higher value indicates better performance of the generated code. 3) *User Requirement Coverage* (URC): we first ask the users to execute the generated code and review the documentation. Next, they are asked to evaluate the extent to which the generated code and doc-

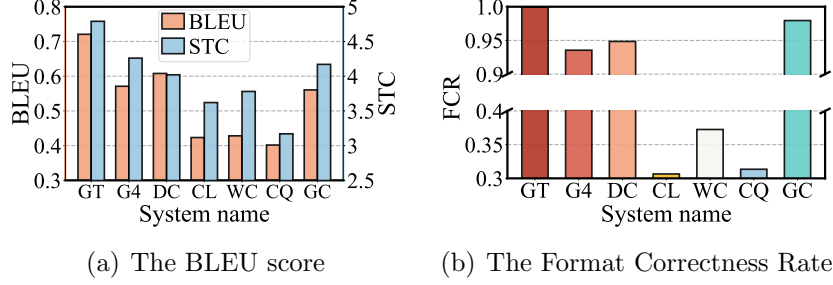


Figure 4.17: Breakdown evaluation on TDSLM.

umentation fulfill all the user requirements. 4) *Code quality*: we assess the quality of the generated code by adopting a commercial-off-the-shelf (COTS) code quality verification tool, SonarQube [123], detecting bug/logic errors, security issues, and code smells [124]. Code smells are not bugs but bad coding styles (*e.g.*, variable name mismatching regular expression) or potential weaknesses (*e.g.*, package version incompatibility). Note that STC and URC are user-related metrics, which are rated on a scale from 1 (not at all) to 5 (completely).

## TDSLM

We input each *problem statement* from *IoT Bench* into TDSLM and the baselines to generate 20 different *decomposed tasks* and calculate the average BLEU score, FCR, and URC. From Fig. 4.17, we observe: 1) The decomposed tasks generated by TDSLM achieve a 48% higher BLEU score than the baselines on average, indicating a stronger decomposition ability for IoT tasks. 2) TDSLM achieves 99% FCR, indicating remarkable stability to generate intermediate output (*decomposed tasks*) based on pre-defined formats. 3) TDSLM also achieves a 28% higher STC on average, showcasing strong abilities in understanding IoT knowledge and generating comprehensive decomposed tasks for IoT applications. Such superior IoT task decomposition and data formatting performance of TDSLM originate from the tuning process on TDD with our IoT-oriented text data augmentation method.

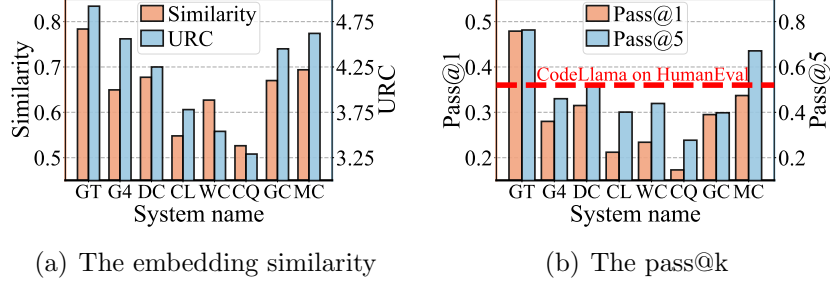


Figure 4.18: Breakdown evaluation on CGSLM

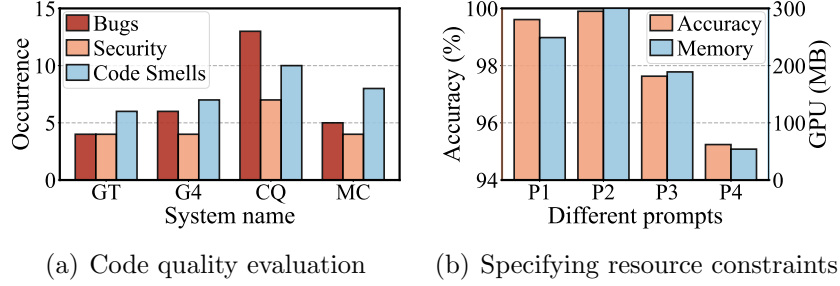


Figure 4.19: (a) Evaluating GPIoT's performance using prompts with different resource constraints. (b) Evaluating the quality of the code generated by CGSLM using SonarQube.

## CGSLM

We input each *task specification* from *IoT Bench* into CGSLM and the baselines to generate 20 different *code & documentation*. We then report the average code embedding similarity, pass@1, pass@5, URC, and the number of various issues detected by SonarQube. by executing and reviewing the code. For comparison, we also report the pass@1 achieved by CodeLlama on a general-purpose programming benchmark, HumanEval [48]. From Fig. 4.18 and Fig. 4.19(a), we observe that: 1) CGSLM achieves an 18% higher code embedding similarity than the baselines on average, implying stronger capability and generalizability in generating IoT-related code snippets. 2) CGSLM achieves higher pass@1 and pass@5 than the baselines, with an average increase of 21.5% and 31%, respectively, showcasing higher quality

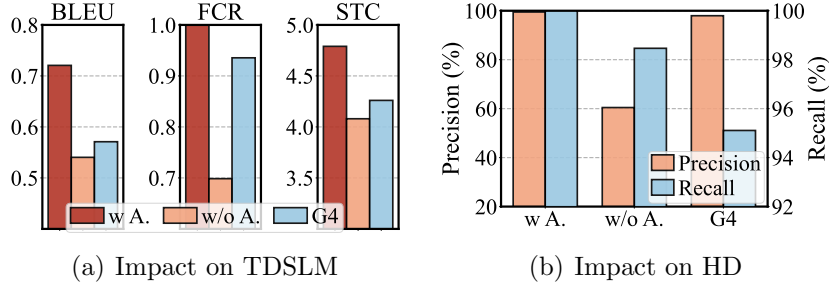


Figure 4.20: Ablation of IoT-oriented augmentation (A.).

and accuracy of the generated code for solving IoT tasks. 3) Nearly all the baselines achieve much lower pass@1 for IoT programming tasks than CodeLlama on HumanEval due to limited capabilities in the IoT domain. 4) Users show a stronger preference for CGSLM over the baselines, with an average increase of 23% in URC. 5) The program synthesized by CGSLM contains fewer bugs, security issues, and code-smell-related issues. This is primarily because, CGSLM, fine-tuned on our manually crafted datasets, can synthesize programs with enhanced code quality. Such superior abilities of CGSLM in generating IoT-related code snippets with high URC stems from our co-tuning on CGD with well-structured data. Consequently, CGSLM can generate code adopting more IoT-specific algorithms following task specifications.

#### 4.6.5 Ablation Study

We conduct an ablation study by removing some proposed technical modules to investigate their importance to GPIoT.

##### IoT-Oriented Data Augmentation.

We directly tune TDSLM on the raw dataset without our IoT-oriented data augmentation, which contains only 273 data samples. We then evaluate the tuned model on *IoT Bench* and report the average BLEU score, FCR, and STC. As shown in Fig. 4.20(a), without our augmentation, the tuned model exhibits a substantial per-

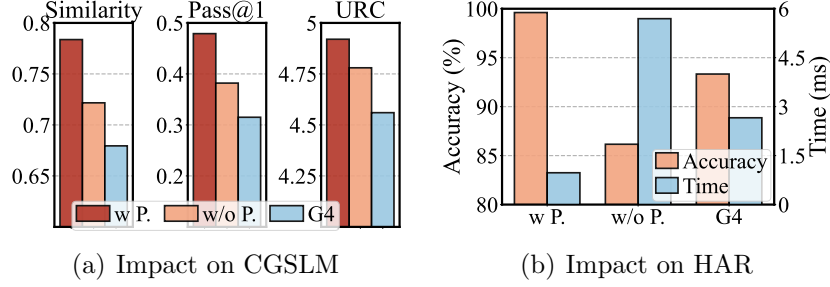


Figure 4.21: Ablation of PECT (P. in the figure).

formance decline across all metrics, much lower than GPT-4o. The main reason is that the raw dataset lacks generalizability and diversity in the IoT domain, which limits the tuned SLM’s ability to decompose IoT problems into manageable sub-tasks, occasionally leading to incorrect results due to hallucinations. As a result, if we use such an insufficiently tuned TDSLM for heartbeat detection, GPIoT still adopts a simple peak detection algorithm, leading to significant performance degradation of the generated code in both precision and recall rate (Fig. 4.20(b)). The results demonstrate the importance of our IoT-oriented text data augmentation method in improving TDSLM’s capability in task decomposition and IoT domain knowledge comprehension.

## PECT

We separately fine-tune TDSLM and CGSLM on their own datasets without our PECT paradigm. We then evaluate the performance of the tuned CGSLM on *IoT-Bench* and measure the average code embedding similarity, pass@1, and URC. As shown in Fig. 4.21(a), without PECT, the code generated by CGSLM exhibits performance degradation across all the metrics. This is because some IoT domain knowledge possessed by TDSLM cannot be shared with CGSLM. As a result, CGSLM cannot handle some programming tasks that are out of the scope, providing simple programs with degraded performance. However, even such insufficiently tuned

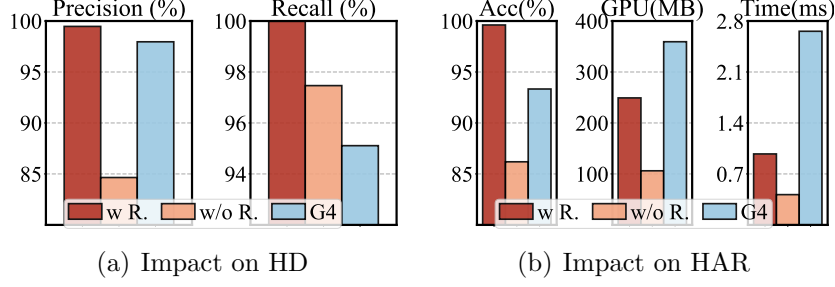


Figure 4.22: Ablation of requirement transformation.

CGSLM still outperforms GPT-4o, highlighting the advantage of fine-tuning in enhancing IoT-related code generation ability. Furthermore, without PECT in the HAR task, the final code neither adopts data pre-processing methods nor designs high-performance neural networks, leading to decreased classification accuracy, as shown in Fig. 4.21(b). The results confirm the importance of our PECT paradigm in mitigating the domain misalignment issue and facilitating the IoT knowledge sharing between TDsLM and CGSLM.

## RTSLM

We directly feed the natural-language-described decomposed tasks from TDsLM into CGSLM. We then compare the performance of the generated code by GPIoT, GPIoT without RTSLM, and GPT-4o. As illustrated in Fig. 4.22, without RTSLM, we find that: 1) In HD, GPIoT tends to generate code unrelated to the IoT domain with substantial precision degradation but a high recall rate. This implies that the heartbeat detection results contain numerous false positives. 2) In HAR, GPIoT designs a basic HAR model with only a few simple layers, leading to a notable accuracy drop. These results highlight the importance of RTSLM in aiding CGSLM to understand the decomposed tasks generated by TDsLM, thereby improving its code generation ability for IoT applications.

#### 4.6.6 User Study

We conduct a user study to evaluate the functionality, generalizability, and overall satisfaction of GPIoT for IoT application development. Specifically, with GPIoT deployed on an edge server, we invite 5 experts and 15 non-experts in IoT and ask them to freely express their requirements for any IoT application development that requires signal processing or AI technologies. By sequentially executing the generated code based on the instructions in the documentation, we ask the users to rate GPIoT based on five metrics: 1) *Overall Code Performance (OCP)* evaluates the overall performance of the generated code on corresponding test data considering task accuracy, runtime efficiency, and resource consumption; 2) *Code & Documentation Readability (CDR)* measures the clarity and structure of the code and documentation; 3) *Generation Efficiency (GE)* assesses how efficiently GPIoT operates in terms of speed and resource usage to produce the final results; 4) *Code Modularity (CM)* judges whether the code is properly modularized for easy reuse and extension; 5) *User Satisfaction (US)* captures users' feedback regarding their overall personal experience. All the above metrics are rated by the users on a scale from 1 (not at all) to 5 (completely). Github Copilot, DeepSeek-Coder (cloud), CodeLlama (local), and MapCoder (agent) serve as representative baselines for comparison.

As shown in Fig. 4.23, we observe: 1) GPIoT significantly outperforms the baselines in terms of OCP and US. The main reason is that, tuned on our IoT-specialized datasets, GPIoT can generate code containing more dedicated algorithms with better performance. Therefore, the users provide a higher score for GPIoT regarding the overall code performance and user satisfaction. 2) GPIoT achieves similar scores to the baselines in terms of CDR and CM, because our datasets mainly focus on generating IoT-related code snippets. Therefore, the readability of the code and documentation are not explicitly enhanced via tuning. 3) GPIoT gets a lower GE score as it performs

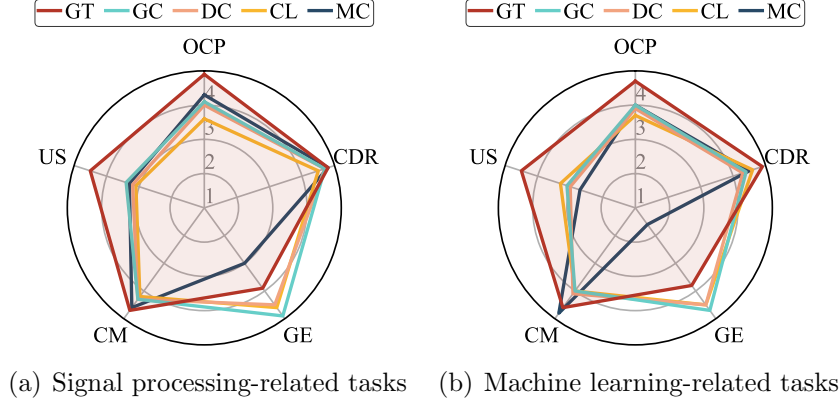


Figure 4.23: User study on different tasks.

requirement transformation and code generation for each decomposed task. Nevertheless, we can enhance its efficiency by adopting various LLM inference and serving optimization methods [186]. Additionally, during our implementation, we find that MapCoder costs around \$20 to synthesize a program for a single IoT application while GPIoT incurs no query costs. The inferior performance of MapCoder may be attributed to the fact that LLM+RAG-based agents typically incorporate multiple modules to generate intermediate results in a cascaded manner, making them susceptible to unstable networks. This yields longer generation time, prohibitive token costs, and degraded user experiences. The user study results demonstrate the superior performance of *GIoT* in synthesizing IoT-related programs in an end-to-end manner, with the ability to generalize to other unseen IoT applications.

## 4.7 Discussion

**System Cost of GPIoT.** In GPIoT, though there are three SLMs (*i.e.*, TDSLM, RT-SLM, and CGSLM) operating simultaneously in GPIoT, they share the same foundation model architecture and only differ in a small subset of tunable parameters (§ 4.4.2). Moreover, though LLM+RAG systems (*e.g.*, MapCoder) require less memory (no more than 1 GB), GPIoT does not need sophisticated prompt design and

requires a shorter generation time for the final output (§ 4.6.6).

**Applicability to Resource-Constrained IoT Devices.** The main focus of GPIoT is generating domain-specific and high-quality code for IoT application development. Considering the resource heterogeneity of various IoT devices for data processing, our *IoT-Oriented Data Augmentation* method (§ 4.4.1) augments the original task decomposition dataset by considering various resource requirements of different target platforms for IoT application development. Further experiments (Fig. 4.19(b)) demonstrate the effectiveness of GPIoT in handling different resource requirements with optimized models.

**Generalizability of GPIoT.** Existing IoT applications can be categorized into four types based on the functionality they deliver: data collection, data transmission, data processing, and decision-making. Data collection, transmission, and decision-making have been extensively studied using fixed programs. For instance, manufacturers typically provide COTS sample code for sensor data collection [187]. In contrast, IoT data processing demands more complex algorithms due to fluctuating sensor data with noises and various resource constraints. Therefore, GPIoT focuses on IoT data processing tasks by offering end-to-end solutions with executable programs. We believe that the general workflow (*i.e.*, task decomposition  $\rightarrow$  requirement transformation  $\rightarrow$  code generation) of GPIoT can be applied to other programming tasks. In future work, we plan to comprehensively assess GPIoT in other complex programming tasks.

## 4.8 Related Work

**Code LLMs & Programming Copilot.** Code LLMs have significantly impacted the field of code generation, with prominent examples such as CodeLlama [49] and DeepSeek-Coder [188]. Trained on vast datasets comprising diverse code repositories,

these models are capable of synthesizing programs based on user requirements, effectively bridging the gap between natural language and code. One notable application is GitHub Copilot, an LLM-powered assistant that provides real-time code suggestions and auto-completion. Though powerful and promising, existing code LLMs and copilots are primarily designed for general-purpose programming, lacking customization to the IoT domain when tasked with IoT applications. GPIoT addresses this by tuning local SLMs on IoT-specialized text-generation datasets with scrupulous augmentation. Additionally, by locally deploying GPIoT, it can potentially serve as a copilot for IoT application developers, enhancing task accuracy and development efficiency in a privacy-preserving manner.

**Data Augmentation for LLMs.** Existing text augmentation methods [189, 190] for LLM tuning harness the advanced language processing capabilities of powerful LLMs (*e.g.*, GPT-4) to synthesize diverse and high-quality text data. These methods have two categories: 1) Depth-based augmentation [157, 191] aims to increase the complexity of the original text data by adding constraints, concretizing the problem, and increasing reasoning steps. 2) Breadth-based augmentation [192] directly uses powerful LLMs to rewrite the original text data and generate a completely new instruction. However, these augmentation methods focus on linguistic characteristics rather than the IoT domain knowledge. In GPIoT, we propose a novel IoT-oriented text augmentation method tailored for the IoT domain, considering unique features of IoT applications, *i.e.*, sensor modalities, data representations, and system resource constraints.

## 4.9 Chapter Summary

We present GPIoT, a tailored local code generation system that synthesizes programs with documentation based on user requirements for IoT application development.

Armed with two IoT-specialized text-generation datasets, the IoT-oriented augmentation method, and our PECT paradigm, **GPIoT** can generate more IoT-related code in a privacy-preserving way, achieving enhanced task accuracy and user satisfaction for IoT application development. As IoT technologies are emerging rapidly, it is also worthwhile to explore the construction of a dynamic IoT knowledge database and continuous fine-tuning of local SLMs in the future.



# Chapter 5

## Fully Autonomous and Adaptive Agentic Workflows for AIoT Development

### 5.1 Introduction

Edge AI integrates AI techniques with heterogeneous mobile devices to enable perception and actuation in real-world environments [8, 193, 194, 195, 196], facilitating various applications such as smart sensing [197, 198, 199], healthcare [200, 201], and autonomous driving [202, 203, 204, 205].

Developing such applications, however, relies on substantial domain expertise of developers, who have to analyze multimodal datasets, design complex signal processing pipelines, train AI models under diverse constraints, and deploy them on mobile devices with heterogeneous runtimes [206]. Consequently, recent works [7, 6] build LLM-based agentic systems to automate this process by leveraging LLMs’ *code generation capabilities* and embedded domain expertise\*. These systems can translate user requirements into executable edge AI programs based on handcrafted, predefined agentic workflows, chaining modules such as knowledge retrieval, task planning, code generation, and iterative refinement [3, 61].

---

\*General coding assistants like GitHub Copilot [41] often achieve suboptimal performance as

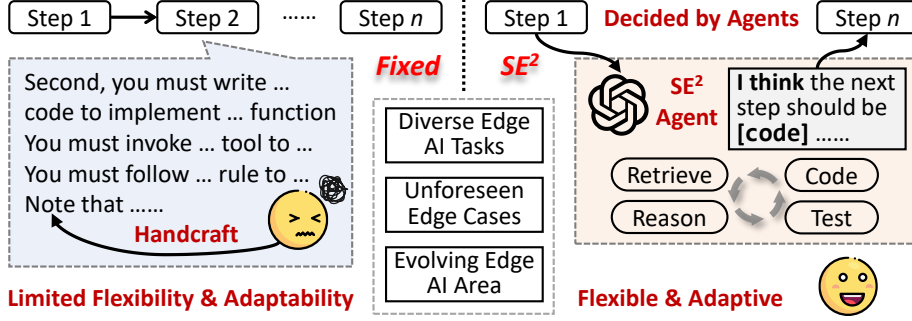


Figure 5.1: Previous works handcrafted fixed agentic workflows to synthesize programs. SE<sup>2</sup> autonomously explores and selects the next atomic operation in program synthesis to meet diverse edge AI requirements.

Fixed agentic workflows, however, often struggle to handle diverse edge AI tasks in real-world scenarios (Fig. 5.1, left). In practice, edge AI tasks may span numerous mobile devices, heterogeneous runtime environments, and dynamic performance and resource constraints. As our study reveals (§ 5.2.1), even well-engineered agentic systems may struggle to accommodate such variability, suffering from *inflexibility*, *limited adaptability*, and *high maintenance costs* [207].

The root cause is that fixed workflows are often *constrained by the specific assumptions and development experiences of workflow designers*. In practice, developers must explicitly enumerate workflow steps and handcraft detailed prompts that regulate agent actions for each. As a result, LLMs solely function as *code-generators* following predefined instruction flows. Yet, a key strength of LLMs remains under-explored: their *reasoning capabilities*, *i.e.*, the ability to make decisions in response to dynamic context information. This gap motivates a key research question: *Can we harness LLMs’ reasoning capabilities to autonomously construct adaptive workflows to meet the ever-changing requirements of edge AI development?* If successful, we could shift the burden of workflow design from developers to AI agents, enabling more flexible, adaptive, and scalable edge AI application development.

they lack domain knowledge of mobile devices [4, 5].

Moving toward such a self-exploring and self-evolving agentic paradigm, we propose  $\text{SE}^2$  that leverages LLMs’ reasoning abilities to autonomously construct distinct workflows for various edge AI tasks. The core insight is inspired by human learning processes, which involve iterative knowledge acquisition, reasoning, practice, and reflection on accumulated experiences. Likewise, an  $\text{SE}^2$  agent explores its workflow and evolves through iterations of reflection and exploration. Building upon this vision, we distill edge AI development into four atomic operations (**retrieve**, **reason**, **code**, **test**), which serve as meta building blocks of agentic workflows (Fig. 5.1, right). The  $\text{SE}^2$  agent autonomously selects the next appropriate meta operation based on its reasoning about the current state and accumulated context, thereby flexibly adapting to diverse edge AI scenarios. In addition, our  $\text{SE}^2$  agent evolves through self-reflection, rather than blindly adhering to rigid workflows.

Nevertheless, three unique technical challenges needed to be addressed, calling for novel design and implementation:

❶ **Limited long-context reasoning capability.** It is inefficient and impractical to rely on a single agent to perform all the atomic operations due to the limited long-context reasoning abilities of current LLMs, even if they have large context windows [208]. To tackle this, we adopt a hierarchical *coordinator-actor agentic architecture*, where a *coordinator* agent maintains a global view and autonomously orchestrates four task-specific atomic agents, *i.e.*, *retriever*, *reasoner*, *programmer*, and *tester*. As we report in detail (§ 5.4), this modular division of cognitive tasks effectively reduces reasoning complexity for agents over long contexts and fosters interpretable and traceable decision-making.

❷ **Lack of in-depth understanding of edge devices.** Our feasibility study (§ 5.2.2) found that, even augmented with detailed device specifications, LLMs still struggle to generate executable and performant edge AI programs. The root cause is that LLMs lack runtime feedback from real-world mobile devices. Consequently,

the generated code often overlooks critical operational semantics and hardware constraints (*e.g.*, CPU architectures, environment affinity, and memory allocation). To address this issue, we propose a *mobile runtime feedback mechanism* that encodes grounded, runtime-level semantics and conveys them to  $\text{SE}^2$  agents as actionable feedback, thereby closing the loop for optimization and iteration.

**③ Inability to adapt to evolving Edge AI tasks.** Simply relying on vanilla LLMs for decision-making cannot cope with the emerging and evolving edge AI environments and requirements. To enhance the adaptability of the *coordinator* as well as the atomic agents, we adopt reinforcement fine-tuning (RFT) [209] with an *ensemble mutual-actor-critic strategy* to improve their domain-specific reasoning capabilities. In addition, we design several continuously self-evolving graders to evaluate the reasoning quality in edge AI contexts and provide adaptive feedback to the RFT loop. This approach enables agents to acquire new domain knowledge over time and agilely adjust their workflows to handle previously unseen scenarios and edge cases.

We fully implement  $\text{SE}^2$  and comprehensively evaluate it against 8 SOTA methods on an edge AI code-generation benchmark and three representative *multimodal* edge AI applications (*human activity recognition*, *vision-language understanding*, and *autonomous driving*), spanning distinct modalities, reasoning complexity, and real-world constraints. Extensive experiments reveal that baseline coding agents cannot consistently generate performant programs (40% success rate lower than  $\text{SE}^2$ ). In contrast,  $\text{SE}^2$  achieves up to 26% higher accuracy and 28% reduction in resource consumption compared to SOTA methods, while retaining superior *generalizability* across tasks. These results validate the effectiveness of our *reasoning-driven* adaptive workflow construction paradigm.

In summary, we make the following contributions:

- $\text{SE}^2$  exploits LLMs’ reasoning capabilities to autonomously construct agentic workflows via self-exploration and self-evolution, addressing the limitations of fixed workflows in heterogeneous and evolving edge AI environments.
- We design a reasoning enhancement framework featuring atomic-operation-level adaptive workflow construction, runtime-grounded probe synthesis, and continual multi-agent reasoning evolution. We also share actionable insights (§ 5.6) for designing future adaptive agentic frameworks in complex, dynamic environments.
- We implement and extensively evaluate  $\text{SE}^2$  against eight SOTA agents on a public benchmark, three representative multimodal edge AI applications, and through a user study, demonstrating its superior accuracy and generalizability.

## 5.2 Background & Motivation

### 5.2.1 Limitations of SOTA Methods

LLMs exhibit exceptional language generation and reasoning abilities. By equipping LLMs with external tools (*e.g.*, search engines, databases), we can build agents [210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221] and design agentic workflows guiding them to automatically solve complex tasks such as edge AI development. As shown in Fig. 5.2, an agentic workflow is typically a multi-step process where each step receives the output from the previous step and produces a new intermediate result. With carefully crafted prompts and tool usage rules in each step, the agent can follow the workflow and synthesize various executable edge AI programs for users.

To better understand the limitations of existing agentic systems and workflows (*i.e.*, *inflexibility*, *limited adaptability*, and *high maintenance costs*), we conduct a preliminary study by reproducing three recent works for edge AI development: AutoIOT

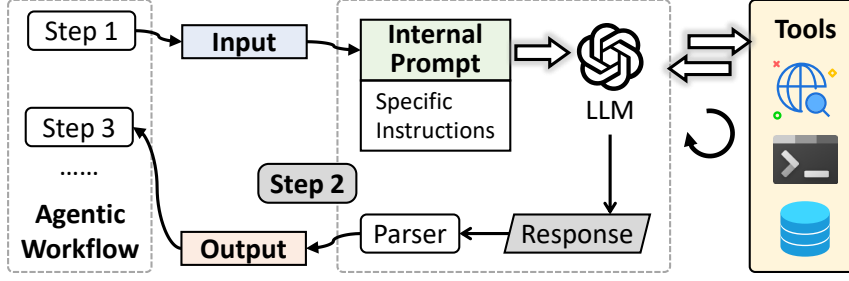


Figure 5.2: A typical agentic workflow of existing work (Step 3 of the workflow is detailed in the figure).

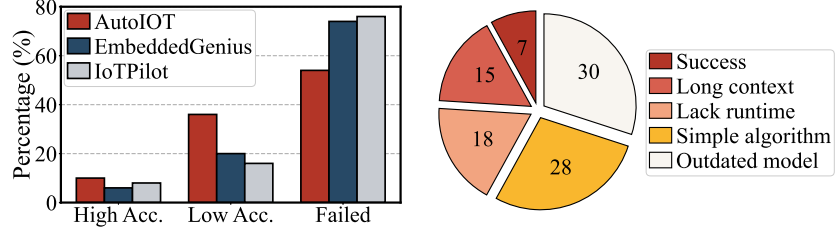
[4], EmbedGenius [7], and IoTPilot [6]. These SOTA methods adopt developer-predefined workflows typically involving background knowledge retrieval, task decomposition, code generation, and debugging.

We request them to develop a human activity recognition system (HAR) using the *multimodal* XRF55 dataset [115] and repeat 100 times to avoid randomness. We randomly remove some data to simulate the modality missing issue that occurs in real-world multimodal scenarios [222]. As shown in Fig. 5.3(a), **only 32% of the synthesized programs of SOTA methods can be successfully executed**, and only 8% achieve comparable performance to a SOTA handcrafted program. **It is highly challenging to design a fixed, one-size-fits-all workflow to adapt to unforeseen cases and accommodate diverse edge AI requirements.**

Further inspection reveals heterogeneous issues:

- *No unimodal verification*: They do not verify the correctness of each code snippet for unimodal data processing before multimodal fusion. This is often discouraged in current multimodal application development pipelines [223].
- *No handling of missing modalities*: 68% of the programs fail when certain data streams are absent due to the lack of elastic AI models for dynamic input handling [222].

To remedy the above as well as other heterogeneous failures, developers have to



(a) Existing agentic systems typi- (b) Though feasible, most agents ex-  
narios cally fail to handle unforeseen sce-  
narios hibernated limited reasoning capabilities

Figure 5.3: Preliminary & feasibility study.

repeatedly revise and refine their workflows. For example, [P.4](#) shows a common prompt template for code snippet generation. Developers must explicitly list detailed regulations, such as tool usage rules, edge-case handling advice, and consistency checks. Maintaining such crafted rules for unseen scenarios with diverse requirements becomes highly time-consuming and error-prone.

#### P. 1: Code Snippet Generation (simplified)

**User Problem:** `{user_problem}` ... `{/user_problem}`

##### Target

According to the detailed design and the context documents, write some Python code to implement `{subtask}`, with the target of `{target}` ...

##### Rules (incomplete)

- For each module, you must use the `{tool_name}` tool multiple times ...
- The code should be consistent with previous snippets, `{code_rules}` ...
- Carefully consider edge cases and failures, including `{edge_cases}` ...

## 5.2.2 Motivation & Feasibility Study

**Motivation.** The root cause of these limitations is that the workflow design and its underlying reasoning still heavily rely on human developers, while LLMs are merely treated as *code generators* following predefined program synthesis pipelines. As a result, these fixed workflows cannot adapt to new or unforeseen tasks in practical real-world scenarios. Unlike SOTA methods, **SE<sup>2</sup>** exploits LLMs' reasoning abilities to enable agents to autonomously optimize and adjust their own workflows in response

to diverse design goals.

We analyze over 30 agent-based code generation systems and find that nearly all edge AI development workflows (Appendix 9.3) are composed of four atomic agentic operations:

- **retrieve:** An agent retrieves relevant domain knowledge from various data sources (*e.g.*, databases, websites).
- **reason:** An agent analyzes the current and past state based on the available information and makes inferences before selecting the next operation. This operation relies on the agent’s reasoning capabilities for decision-making.
- **code:** An agent generates a Python<sup>†</sup>code snippet.
- **test:** An agent generates test cases, deploys the code snippet on the target mobile device (optional), executes and evaluates it, and records the evaluation results.

**By enabling AI agents to autonomously select among these operations in response to changing states, agentic workflows can be organically constructed.** Similar to next token prediction in LLMs, SE<sup>2</sup> instead predicts the next atomic operation that drives the workflow forward.

**Feasibility & Challenges.** We build an agent to test whether it can construct workflows autonomously. We instruct the agent to synthesize a multimodal HAR program and run it on Jetson Nano by iteratively selecting and executing atomic operations until it satisfies the requirement.

We repeat the test 100 times to avoid randomness. Results show that certain constructed workflows can sometimes generate executable edge AI programs with satisfactory performance after multiple trials. Notably, they share a similar and

---

<sup>†</sup>We focus on Python since it is widely used for AI development [224].

commonly adopted structure: “retrieve knowledge → generate code snippets → test individually → multimodal fusion → build elastic AI models → iteratively refining”. These results imply that adaptive workflow is feasible. Despite this feasibility, the overall success rate remains low (8/100), comparable to directly running existing systems without modification. Further in-depth analysis reveals that the failures stem from three fundamental reasoning limitations (Fig. 5.3(b)), which precisely motivate the technical challenges that SE<sup>2</sup> is designed to overcome:

- **Long-Context Reasoning.** 16 agents repeatedly execute the same action (*e.g.*, searching dataset information) and fail to terminate. It is likely because a single agent performs all operations and accumulates increasingly long contexts, while LLMs inherently pay more attention to the earliest and most recent tokens in the context window, causing the agent to “forget” previously executed steps [208].
- **Runtime-Aware Reasoning.** Even with retrieved domain knowledge of the target device, 18 agents still produce unexecutable programs either because their required runtime environments are incompatible with the target device or because their resource consumption exceeds the device’s capacity. The reason is that the agent cannot access the target device and thus lacks an in-depth understanding of its runtime environments and resource constraints.
- **Outdated Models.** 58 agents adopt overly general or outdated algorithms and AI models. This indicates that LLMs lack the reasoning capabilities to associate user requirements with SOTA edge AI algorithms and models.

**Remarks.** These findings highlight that, while autonomous workflow exploration is possible, current LLMs’ reasoning capabilities fall short in meeting our requirement of fully automating program synthesis workflows.

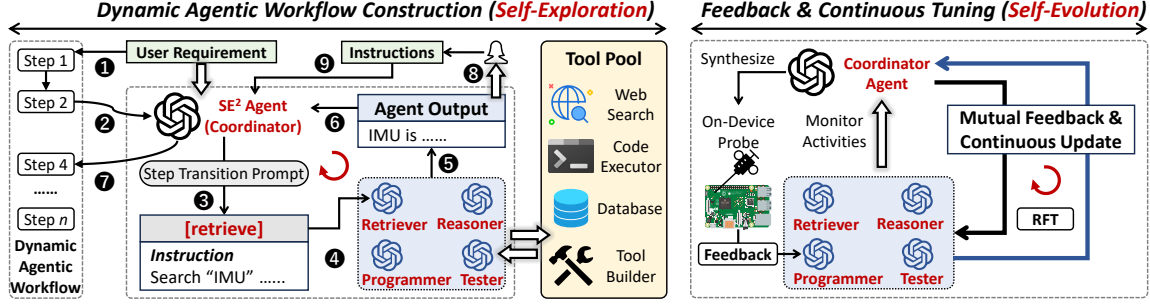


Figure 5.4: The overall architecture and workflow of  $SE^2$  (with detailed operations in step 3).

To address these challenges, we design and implement a series of novel techniques: a hierarchical *coordinator-actor agentic architecture* (§ 5.4.1), a *mobile runtime feedback mechanism* (§ 5.4.2), and a *continuous LLM tuning paradigm* (§ 5.4.3).

### 5.3 $SE^2$ Overview

The goal of  $SE^2$  is to enhance and exploit LLMs’ reasoning capabilities to dynamically and adaptively construct agentic workflows, which translate user requirements into edge AI programs executable on diverse mobile devices. We adopt **retrieve-reason-code-test** as four atomic operations, serving as minimal cognitive units of agentic workflows. This abstraction provides sufficient expressiveness for diverse pipelines while keeping the *coordinator*’s decision space small enough for reliable reasoning. As shown in Fig. 5.4,  $SE^2$  consists of two main stages: 1) *self-exploration* for dynamic workflow construction, and 2) *self-evolution* for runtime feedback and continuous LLM tuning.

**Self-Exploration.** Given a user requirement for edge AI development (❶), the *coordinator* iteratively determines the next appropriate operation. At each step, the *coordinator* will analyze the output from the previous operation (❷) using a step transition prompt, and generate the next atomic operation along with detailed instructions (❸). The corresponding atomic agent is then activated (❹) to execute the

operation based on the instruction and generate an output (❸). The output is further sent to the *coordinator* (❹). Based on the currently constructed workflow, the *coordinator* decides if extra operations are required (❺) or concludes that the synthesized edge AI program meets the user requirement with acceptable performance (❻). In addition,  $SE^2$  provides an interface for users to offer specific instructions for workflow refinement (❼). As such, agentic workflows are dynamically and autonomously determined with minimal user intervention, offering enhanced flexibility and adaptability. Note that to better interpret and verify user intents, we can orthogonally integrate existing arts (*e.g.*, LLM-as-a-Judge [225]).

**Self-Evolution.** During workflow construction, when the *tester* runs the generated code on the target device, our *mobile runtime feedback mechanism* synthesizes a probe to extract runtime semantics (*e.g.*, memory allocation) as grounded, runtime-level feedback. In parallel, our *continuous LLM tuning paradigm* records the mutual feedback between the *coordinator* and atomic agents to incrementally enhance their reasoning capabilities in the edge AI domain. As such, all agents can continuously evolve, enabling them to adapt to unforeseen scenarios and emerging edge AI knowledge.

## 5.4 Detailed Design of $SE^2$

### 5.4.1 Coordinator-Actor Agentic Architecture

This module performs **self-exploration** by dynamically constructing adaptive agentic workflows. Due to LLMs’ limited long-context reasoning capabilities (§ 5.2.2), we propose a hierarchical *coordinator-actor agentic architecture*, where a *coordinator* agent orchestrates the self-exploring process by delegating distinct tasks to four atomic agents: *retriever*, *reasoner*, *programmer*, and *tester*. This design offloads the burden of atomic operation execution from a single agent to multiple task-specific

agents for more efficient handling.

**Challenge.** In multi-agent systems, sharing contexts among agents often leads to prohibitive costs [226], introducing a new challenge: *how to enable effective and efficient communication between the coordinator and atomic agents while ensuring context coherence?* To address this, we design tailored prompts to regulate each agent’s behavior and outputs.

**Coordinator Prompt** (P.5) for step transition includes:

- *Role & objective.* We assign the agent a *coordinator* role and provide a task description that guides it to proactively orchestrate multiple agents for edge AI development.
- *Respond rules.* We provide detailed descriptions of the four atomic operations and their selection rules. We task the *coordinator* with iteratively selecting an appropriate operation and activating the corresponding atomic agent for execution. We also introduce a **quit** operation that ends self-exploration if the synthesized program fulfills the user requirement. Importantly, beyond sticking to the general cycle of “*retrieve*  $\rightarrow$  *reason*  $\rightarrow$  *code*  $\rightarrow$  *test*”, we encourage the *coordinator* to repeat any stage as needed, *e.g.*, “*retrieve*  $\rightarrow$  *reason*  $\rightarrow$  *retrieve*”. As such, the *coordinator* can dynamically explore workflows for various edge AI tasks.
- *Current state.* Atomic agents’ activities are recorded during operation execution, serving as the current state and thereby enabling the *coordinator* to determine the next operation with enhanced contextual coherence.
- *Output specifications.* To ensure that atomic agents can accurately interpret the operation, we stipulate that the *coordinator* only outputs three items: the operation name, a brief instruction, and relevant context information.

- *One-shot example.* We provide a sample output to reduce ambiguity in task interpretation.

**P. 2: Coordinator Prompt (simplified)**

❶ **Role & Objective:** You are a **coordinator** agent operating in a multi-agent system for edge AI development. Given a **user requirement**, you need to orchestrate another four atomic agents (**retriever**, **reasoner**, **programmer**, and **tester**) to synthesize an executable program.

❷ **Respond Rules:** You **MUST** select one of the following operations:

1. **Retrieve** – activate the **retriever** to retrieve relevant knowledge ...
2. **Reason** – activate the **reasoner** to analyze and reflect on ...
3. **Code** – activate the **programmer** to write a Python script ...
4. **Test** – activate the **tester** to execute and evaluate the code ...
5. **Quit** – end the task if the program satisfies the user requirement.

❸ **Current State:** The summarized activity of **[atomic agent]**: {.....}

❹ **Output Specifications:** You **MUST** generate **ONLY THREE** items:

1. **Operation Name** – the name of the atomic operation for execution.
2. **Instructions** – a brief instruction for the corresponding agent ...
3. **Context** – detailed context including previously generated ...

❺ **Simple Example Response**

[retrieve] Retrieve detailed information of the {dataset name} dataset.  
I need to understand the dataset first, including dimensionality, .....

**Atomic Agent Prompt (P.6)** comprises four parts:

- *Role & objective.* We also assign distinct roles to the atomic agents, along with descriptions of their jobs.
- *Operation execution rules.* We encourage each atomic agent to proactively and repeatedly invoke relevant tools to obtain sufficient information or conduct thorough tests.
- *Output specifications.* We stipulate that each atomic agent can only respond with a **function\_call** with parameters to invoke a tool, or a **message** to quit the current operation.
- *Instruction & context information* from the *coordinator*.

**P. 3: Atomic Agent Prompt (simplified) (Example: *retriever*)**

❶ **Role & Objective:** You are a **retriever** agent operating in ... Based on the provided information, you need to retrieve relevant information from external data sources (*e.g.*, websites).

❷ **Operation Execution Rules:** You **MUST** follow these rules:

1. You **MUST** invoke multiple tools to complete the instruction ...
2. You **MUST** retrieve as much information as possible before ...
3. You **MUST** summarize the retrieved content and store it ...

❸ **Output Specifications:** Your **MUST** output one of the types below:

1. *function\_call* – you need to invoke tools to gather information or ...
2. *message* – directly output “[finish\_retrieval]” if you believe that ...

❹ **Instruction & Context Information:**

{*coordinator\_instruction*}    {*context\_information*}

We instruct atomic agents to invoke tools until all necessary information is gathered. Once a “finished” **message** is generated, the atomic agent’s activity is summarized and returned to the *coordinator* for further decision-making.

**Remarks.** Through our *coordinator-actor agentic architecture* with crafted prompts, the *coordinator* can autonomously explore appropriate atomic operations based on its reasoning abilities. As such, agentic workflows can be dynamically constructed to cope with unforeseen cases (*e.g.*, modality missing), exhibiting enhanced flexibility and adaptability.

## 5.4.2 Mobile Runtime Feedback Mechanism

This module provides runtime-level feedback to  $SE^2$  for decision adjustment, thus facilitating **self-evolution**. As reported in § 5.2.2, even with retrieved device specifications, some synthesized programs remain unexecutable, as  $SE^2$  does not have direct access to mobile devices, lacking detailed runtime information during coding and testing. Accordingly, we propose a *mobile runtime feedback mechanism* that obtains grounded, runtime-level semantics and delivers them to atomic agents as timely feedback.

**Challenge & Solution.** An intuitive way to extract runtime semantics is to directly apply existing device profiling techniques, such as nn-Meter [227]. However, they

rely on rule-based scripts for runtime monitoring, which are often insufficient for diverse and rapidly evolving edge AI tasks [228]. Rather than building a one-size-fits-all runtime monitor, we propose synthesizing *probe programs* that execute on the target device to extract only relevant runtime semantics. Our insight is that the *coordinator*, overseeing the entire development pipeline, can serve as a supervisor for probe synthesis. The runtime semantics collected by the probe are then fed back to atomic agents for response adjustments. We design three modules for the feedback mechanism: *probe synthesis*, *semantic alignment grading*, and *response adjustment*.

**Probe Synthesis.** Different from general code generation, the purpose of a probe is not to solve the coding task, but to extract latent runtime semantics that agents cannot infer from textual device specifications. Specifically, we prompt (P.7) the *coordinator* to synthesize a probe program with two main considerations before on-device code execution:

- *Runtime semantics.* The four dimensions in the prompt capture runtime properties that static device specs cannot reliably encode. *Environment affinity* reveals incompatibilities between library versions and hardware kernels. *Memory dynamics* reflect feasibility under real allocation and fragmentation patterns. *Execution behavior* exposes performance bottlenecks arising from scheduling, warm-up, and accelerator utilization. *I/O feasibility* verifies if essential data formats are supported.
- *Probe program requirements.* *Minimality* ensures that each probe focuses on a single runtime factor, reducing confounding effects and enabling precise interpretation of observed behavior. *Orthogonality* prevents probes from blending unrelated concerns, preserving the diagnostic clarity needed for targeted adjustments. *Executability* guarantees that each synthesized probe can reliably run within the device’s native software stack and provide grounded evidence.

Finally, *observability* mandates that probes emit clean, structured outputs, enabling the system to consistently extract useful runtime semantics.

**P. 4: Probe Synthesis Prompt (detailed on our website)**

❶ **Role & Objective:** You are a **probe-synthesizer**. Your goal is to develop a probe program that, when executed on the target device, extracts runtime semantics directly relevant to the current subtask.

❷ **Runtime Semantics.** (tailored to the *coordinator*’s instruction):

1. *Environment affinity* (e.g., PyTorch versions, CUDA/cuDNN bindings)
2. *Memory dynamics* (e.g., allocation failure, fragmentation, peak usage)
3. *Execution behavior* (e.g., latency, warm-up stability)
4. *I/O feasibility* (e.g., data loading, sensor interface availability)

❸ **Probe Program Requirements:**

1. *Minimality*; 2. *Orthogonality*; 3. *Executability*; 4. *Observability*

❹ **Instruction & Context Information:**

{coordinator\_instruction}    {context\_information}

**Semantic Alignment Grading.** Based on the runtime semantics obtained from the probe, we need to judge if each atomic agent’s decision aligns with the current device runtime and the edge AI development requirement. Such semantic alignment manifests in two aspects: 1) *Local alignment*: whether the atomic agent’s response contributes to the agent’s current task (i.e., the *coordinator*’s instruction); 2) *Global alignment*: whether it contributes to the *coordinator*’s overarching task (i.e., the user requirement). We prompt the *coordinator* to assign two scores reflecting the local and global alignment of each atomic agent’s response based on six designed grading levels (details in § 9.5). In addition, we instruct the *coordinator* to provide feedback for atomic agents to adjust their responses. For example, “The code should use a smaller batch size to reduce memory usage and employ TensorRT to accelerate on-device inference”. As such, the grading process for response  $r_i$  is expressed as:

$$\{s_l^i, f_l^i, s_g^i, f_g^i\} = \mathcal{L}(r_i \mid \mathcal{I}_i, \mathcal{A}_i, u, \mathcal{S}) \quad (5.1)$$

where  $s_l^i$  and  $s_g^i$  are the local and global semantic scores,  $f_l^i$  and  $f_g^i$  are the corresponding feedback, respectively.  $\mathcal{I}_i$  is the *coordinator*’s instruction.  $\mathcal{A}_i$  is the atomic

agent’s activity record during operation execution.  $u$  is the user requirement.  $\mathcal{S}$  is the runtime semantics obtained from the probe. The scores and dialogue will be stored in a dataset  $D_a$ :

$$D_a = \{(\mathcal{A}_1, s_l^1, s_g^1), (\mathcal{A}_2, s_l^2, s_g^2), \dots\}, \quad (5.2)$$

which will be further used for continuous tuning (§ 5.4.3).

**Response Adjustment.** Considering both the current state and the feedback from the *coordinator*, we prompt the atomic agent to adjust its response and regenerate a better one:  $\hat{r}_i = \mathcal{L}(\mathcal{A}_i \mid (f_l^i, f_g^i))$ . Note that we do not request agents to regenerate if the semantic score exceeds four, as the response is sufficiently aligned with the intended task, exhibiting only minor flaws that do not impact overall coherence [229].

**Remarks.** We leverage the *coordinator* to provide guidance grounded in mobile devices’ runtime constraints, enabling atomic agents to adjust responses that better align with both the current operation and the overall user requirement.

### 5.4.3 Continuous LLM Tuning Paradigm

This module performs **self-evolution** by continuously fine-tuning both the *coordinator* and atomic agents to enhance their reasoning in comprehending evolving edge AI knowledge and tasks. Specifically, we focus on three aspects: 1) The *coordinator* should exhibit enhanced cognitive reasoning to autonomously determine the next appropriate atomic operation based on its current state. 2) Each atomic agent should execute its operation properly. 3) The *programmer* should possess domain-specific reasoning capabilities to interpret edge AI knowledge and generate relevant algorithms.

To achieve these goals, we can intuitively build a continuously updated RAG database [230]. Yet, timely maintenance demands substantial domain expertise, and

vanilla LLMs still require strong reasoning abilities to comprehend the new knowledge [5]. In contrast, we adopt a *modified* reinforcement fine-tuning (RFT) [209] strategy to enhance agents’ reasoning abilities in the edge AI domain without user intervention. Different from supervised fine-tuning [231, 232, 233], RFT can enhance LLMs’ reasoning capabilities in specific domains by tuning on smaller amounts of data [209].

**RFT Procedure.** Rather than forcing LLMs to generate precise answers, RFT relies on a grader that scores generated responses, followed by policy gradients to refine LLM parameters [234]. RFT mainly consists of three stages: 1) *Grader design*. A grader is created to score the quality of LLM outputs. Generally, graders can take various forms, such as format checkers, Python code, or even LLMs [209]. 2) *Tuning dataset construction*. We first prepare a question set,  $Q$ , with an expected answer set,  $A_e$ , and input  $Q$  into the LLM,  $\mathcal{L}$ , to obtain a generated answer set,  $A_g$ . Accordingly, we prompt the grader,  $\mathcal{G}$ , to score each answer. The tuning dataset,  $T$ , is thus obtained as a set of questions, expected answers, generated answers, and scores,  $s_i$ . The entire process is:

$$\begin{aligned} a_i^g &= \mathcal{L}(q_i), \quad \forall q_i \in Q \\ s_i &= \mathcal{G}(q_i, a_i^g, a_i^e), \quad \forall a_i^g \in A_g, \forall a_i^e \in A_e \\ T &= \{(q_i, a_i^g, a_i^e, s_i), \dots\} \end{aligned} \quad (5.3)$$

3) *Fine-tuning*. With the obtained training dataset, policy-gradient updates (*e.g.*, VPG [234]) are adopted for LLM tuning:

$$\hat{g} = \frac{1}{|T|} \sum_{s_i \in T} \nabla_{\theta} \log s_i \big|_{\theta} \hat{A}, \quad \theta' = \theta + \eta \cdot \hat{g} \quad (5.4)$$

where  $\hat{g}$  is the policy gradient,  $|T|$  is the dataset cardinality,  $\theta$  are LLM parameters,  $\hat{A}$  are the advantage estimates [235] based on the grader, and  $\eta$  is the learning rate. After several epochs, RFT enhances the reasoning capabilities of LLMs by

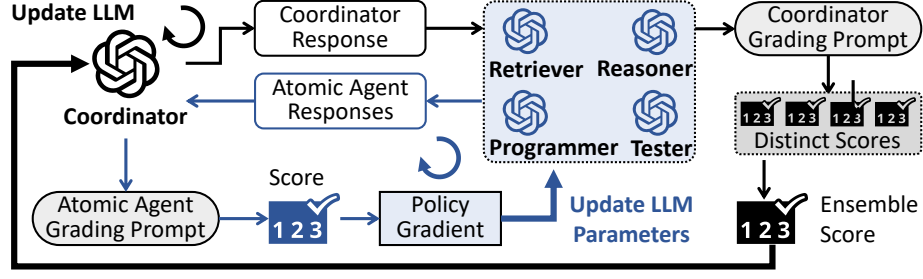


Figure 5.5: Ensemble mutual-actor-critic strategy.

encouraging them to generate high-scoring outputs more frequently while suppressing low-scoring ones.

**Challenges.** Directly adopting traditional RFT in  $SE^2$  faces two main challenges: 1) As agents continuously evolve, we also need to design multiple evolving graders to assess both the *coordinator* and atomic agents. 2) Grading the *programmer* demands complicated analysis of both the quality and performance of the generated code across diverse edge AI algorithms, calling for systematic feedback.

**Solutions.** To tackle these issues, we propose: 1) An *ensemble mutual-actor-critic strategy* to enhance both the *coordinator* and atomic agents in a mutual-critique manner. Our insight is that both the *coordinator* and atomic agents can function as graders to criticize each other’s behaviors from distinct perspectives. 2) A *self-reflective grader* to further critique the *programmer*, considering both code performance and algorithm specialization to the edge AI domain.

**Ensemble Mutual-Actor-Critic Strategy.** We augment the actor-critic algorithm [236] with an ensemble updating strategy to refine both the *coordinator* and atomic agents continuously. As shown in Fig. 5.5, to update atomic agents, we treat the *coordinator* as the critic (*i.e.*, grader) to evaluate atomic agent behaviors. In turn, to refine the *coordinator*, we regard atomic agents as four separate graders, each criticizing the *coordinator* from a distinct angle regarding their specific tasks. Thus, both the *coordinator* and atomic agents can be mutually updated as self-

exploration progresses. Specifically, given the dataset  $D_a$  in Equ. 5.2 that contains the scores for the atomic agents, their tuning processes are expressed as:

$$s_i^a = \alpha \cdot s_i^l + \beta \cdot s_i^g, \hat{g} = \frac{1}{|D_a|} \sum \nabla_{\theta} \log s_i^a|_{\theta} \hat{A}, \theta' = \theta + \eta \cdot \hat{g} \quad (5.5)$$

where  $s_i^a$  is a linear combination of both local and global semantic scores graded by the *coordinator*. To tune the *coordinator*, we first design four tailored grader prompts for atomic agents to rate the current atomic operation and instruction generated by the *coordinator*. Taking the *retriever* as an example, the grading prompt (P.8) considers both the correctness and necessity of the *coordinator*'s decision. An ensemble score is then calculated via weighted aggregation:

$$\begin{aligned} s_i^c &= \sum_{o \in \mathcal{O}} w_o \cdot \rho_o, \quad s_i^c \in [0, 5] \\ \rho_o &= \begin{cases} s_i^o & \text{if the } \textit{coordinator}'\text{s operation} = o \\ 5 - s_i^o & \text{otherwise} \end{cases} \end{aligned} \quad (5.6)$$

where  $o$  is the current operation,  $w_o$  is its assigned weight, and  $s_i^o$  is the score rated by each atomic agent. If  $o$  does not match the atomic agent's operation, we compute the score as  $5 - s_i^o$ , as in this case, a higher score means that the atomic agent considers its own operation to be more appropriate. The *coordinator* is then updated using the ensemble score:

$$\hat{g} = \frac{1}{|D_c|} \sum \nabla_{\theta} \log s_i^c|_{\theta} \hat{A}, \quad \theta' = \theta + \eta \cdot \hat{g} \quad (5.7)$$

Such an ensemble strategy allows bidirectional critique between the *coordinator* and atomic agents during self-exploration, thereby fostering reflection on accumulated context and enabling continuous, iterative self-evolution.

**P. 5: Coordinator Grading Prompt – by retriever (simplified)**

Given the **coordinator**’s response (the atomic operation and the instruction), please rate it based on the following two questions:

1. If the atomic operation is "retrieve", to what extent does it align with the current context? For example, if the *coordinator* proposes retrieving the same information (e.g., dataset specs) again, the score may be lower; otherwise, it may be higher if the information is important.
2. If not, how likely is it that this should be a "retrieve" operation?

**Self-Reflective Grader for Edge AI.** To further enhance the *programmer*, we consider three main aspects:

- *Code quality* includes syntax correctness and code smells, which are bad coding styles (e.g., no exception handlers) or potential weaknesses (e.g., deprecated packages) [237]. They can be scored by SOTA code quality inspectors.
- *Code performance* is tested by prompting the *tester* to produce test cases and assess performance across multiple evaluation metrics. The *tester* is further prompted to assign a score to comprehensively reflect the results.
- *Algorithm specialization*: Whether the *programmer* uses tailored algorithms to process sensor data. For instance, Butterworth filtering is commonly applied to Wi-Fi CSI data to suppress noise induced by non-human activities.

We prompt the *programmer* to first identify a list of sensor data processing algorithms used in the generated code. For each algorithm, we prompt (P.9) the *programmer* itself to grade it on a scale from 1 (completely mismatched) to 6 (fully tailored) based on how closely the algorithm aligns with established, modality-specific methods. Higher scores indicate stronger adherence to domain-relevant practices, and lower scores reflect reliance on generic or unsuitable approaches. This process exploits the *self-reflection* capability of the *programmer*, with the reflection score further used in RFT to facilitate self-evolution. Then the final score is:

$$s_i^p = s_i^q + s_i^{cp} + \sum \lambda_j \cdot s_i^{s_j}, \quad (5.8)$$

where  $s_i^q$  is the code quality score,  $s_i^{cp}$  is the code performance score,  $s_i^{sj}$  is the  $j$ -th algorithm specialization score with its corresponding weight  $\lambda_j$ .

#### P. 6: Algorithm Specialization Grading Prompt (simplified)

You are a **code auditor** tasked with scoring how well the algorithm is tailored to the target data modality, rather than using generic methods.

##### **Tailoring score (16):**

- 6 – *Fully*: Domain algorithms with modality-appropriate preprocessing.
- 5 – *Mostly*: Domain algorithm present with minor omissions.
- 4 – *Partially*: Some domain steps present, but key elements are missing.
- 3 – *Weakly*: Mostly generic with possible accuracy degradation.
- 2 – *Not tailored*: Generic algorithm; lacks essential modality logic.
- 1 – *Mismatched/harmful*: Algorithm contradicts modality requirements.

**Remarks.** Our *continuous tuning paradigm* enables mutual critique between the *coordinator* and atomic agents for continual reasoning enhancement, while further improving the *programmer* through self-reflection. Such bidirectional supervision and collaborative progress mitigate LLM hallucinations and foster a more robust self-evolution paradigm for adaptive agentic systems in edge AI development.

## 5.5 Experiment Setup

### 5.5.1 Implementation

We fully implement the agentic framework **SE**<sup>2</sup>. The default LLM is OpenAI’s o4-mini [238] for balanced reasoning and response latency. The initial tool pool contains a search engine based on Tavily [98], an empty database based on Chroma [239], a Python executor inside a sandbox, and a tool builder that allows agents to dynamically create new tools. We use Pylint[240] for code quality inspection. *We carefully design and implement all graders and adopt RFT following OpenAI’s official guidelines* [209]. We develop **SE**<sup>2</sup> with 5,288 lines of Python code and deploy it on a local server. The synthesized programs are tested and deployed on various mobile devices, including portable laptops, NVIDIA Jetson Nano, and Raspberry Pis, which are

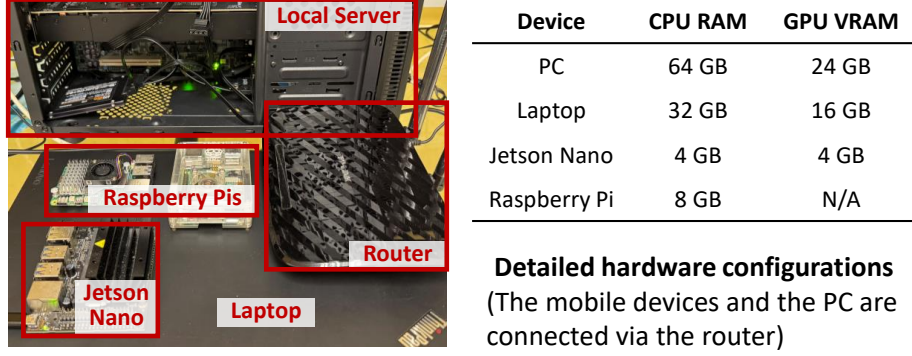


Figure 5.6: Real-world experiment setup.

connected to the local server via a router. The hardware configurations are detailed in Fig. 5.6.

### 5.5.2 Evaluation Protocols

We use both a SOTA benchmark and three representative edge AI applications for comprehensive evaluations. The benchmark provides a controlled setting for measuring core capabilities, while the real-world edge AI cases demonstrate the system’s effectiveness and adaptability in practical scenarios. In addition, we also conduct a user study to evaluate the *generalizability* of SE<sup>2</sup>, where we invite users to propose diverse improvisational requirements (including unseen applications) for edge AI development.

#### Benchmark

To comprehensively evaluate the overall edge AI development abilities of SE<sup>2</sup>, we adopt *IoTBenCh* [5], a benchmark specifically designed to assess LLMs’ abilities in generating IoT-relevant programs. *We filter out data irrelevant to the edge AI domain from the benchmark and augment the remaining samples to 200 coding tasks following the methodology in [5].* Each task consists of a “*task specification* → *edge AI code*” pair with curated functional test cases, each corresponding to a specific func-

tional requirement in the specification. We randomly split the augmented benchmark into two parts: 20% for tuning and 80% for evaluation (RFT requires less tuning data). *The tuning and testing data have completely different edge AI tasks.*

## Multimodal Edge AI Applications

Considering different *developing pipelines, data processing algorithms, sensor modalities, and device heterogeneity*, we select three representative multimodal edge AI applications for evaluation:

❶ **Human Activity Recognition (HAR)** using the XRF55 dataset [115], containing 42.9K samples and 55 action classes of various human daily activities. Three modalities are involved: RFID, Wi-Fi CSI, and mmWave. *We request deployment on a Raspberry Pi with no GPU and only 8GB RAM.*

❷ **Vision-Language Understanding (VLU)** using the MMMU dataset [241] for image Q&A. There are two modalities in the dataset: image and text. *We request deployment on a Jetson Nano, which has a different runtime environment from the local server where  $SE^2$  operates.*

❸ **Object Detection in Autonomous Driving (ODAD)** using the aiMotive dataset [242], comprising 176 scenes with calibrated data. This task is the most complex one as it involves four modalities (1 LiDAR, 2 radars, 4 cameras, and 1 GNSS+INS) with high dimensionality, necessitating complex data processing algorithms and AI models. *Besides, we randomly remove some data modalities to simulate the modality missing issue in real-world scenarios.*

## Baselines.

We compare  $SE^2$  with eight SOTA baselines based on predefined workflows (detailed in § 9.3.1), which can be categorized into the following three types:

- **General coding LLM:** *GitHub Copilot* [41] suggests code snippets and func-

tions following user instructions.

- **General coding agents:** *MapCoder* [1], *AgentCoder* [2], and *AgentSquare* [3].
- **Coding agents tailored to mobile tasks:** *AutoIOT* [4], *GPIoT* [5], *IoTPilot* [6], and *EmbedGenius* [7].

## 5.6 Evaluation

### 5.6.1 Benchmark Evaluation

#### Metrics

1) *Code similarity*: we convert code into embeddings that capture its internal logic via CodeT5+ [185]. We compute the cosine similarity between the extracted internal logic embedding of the generated and the reference code. A higher value indicates a stronger ability to generate edge-AI-related code. 2) *Pass@k*: measures the probability that at least one of the top-k generated programs passes all functional test cases [48]. A higher value indicates SE<sup>2</sup> accurately interprets task specifications and thus passes more functional test cases from IoTBench. 3) *Code quality*: we use Pylint to assess code quality with a full score of ten.

#### Performance Comparison

Using the augmented *IoT Bench*, we present the recorded metrics in Fig. 5.7 and find:

**SE<sup>2</sup> can effectively generate functionally correct and performant code tailored to edge AI following task specifications.** On average, SE<sup>2</sup> achieves an 11% higher code similarity score and a 17% higher pass@1 rate than baselines. The gain in pass@5 rate (10% ↑) is relatively small because it has higher fault tolerance than pass@1. Furthermore, we find that compared with the baselines, the standard deviations of SE<sup>2</sup> across all metrics are much lower than the baselines, achieving 0.05,

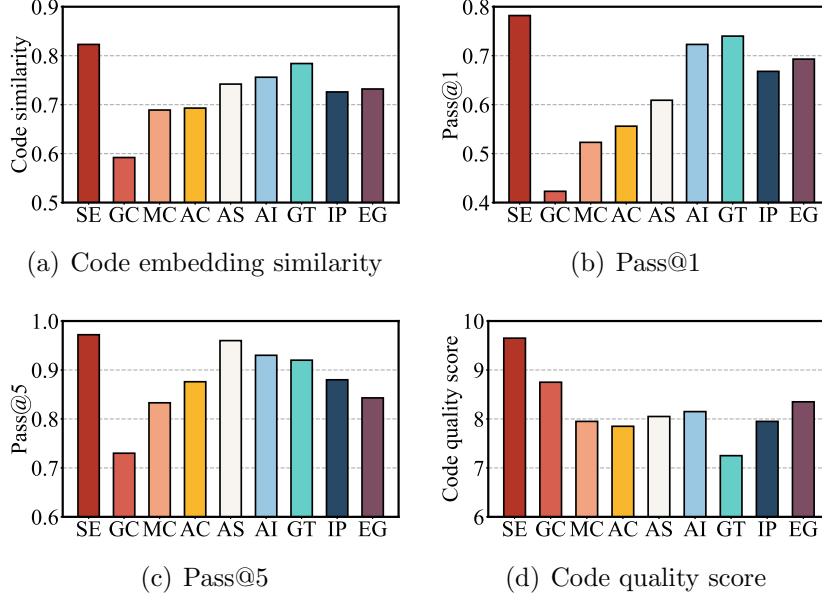


Figure 5.7: Performance comparison on *IoT Bench* (SE:  $SE^2$ , GC: GitHub Copilot, MC: MapCoder, AC: AgentCoder, AS: AgentSquare, AI: AutoIOT, GT: GPIoT, IP: IoTPilot, EG: EmbedGenius).

0.04, 0.02, and 0.73 in terms of code similarity, pass@1, pass@5, and code quality score, respectively.

Interestingly, general coding agents (GitHub Copilot, MapCoder, AgentCoder) exhibit lower performance in terms of code similarity (14% ↓) and pass@1 rate (25% ↓), indicating that general coding agents are incapable of generating high-quality edge AI programs. For instance, they all directly apply AI models on the raw Wi-Fi CSI data without dedicated processing (*e.g.*, Butterworth filtering [40]) to remove noise induced by non-human activities in HAR.

In contrast, IoT-tailored agents (AutoIOT, GPIoT, IoTCopilot, EmbedGenius) perform relatively better, owing to tailored designs such as RAG and fine-tuning. In addition, AgentSquare achieves comparable performance to  $SE^2$  in terms of pass@5 rate due to its iterative workflow optimization method for performance maximization. However, a further study (§ 5.6.2) shows that it requires

significantly more time to synthesize performant programs.

**SE<sup>2</sup> significantly outperforms the baselines in code quality by an average of 1.6 points.** The underlying reason is that our *self-reflective grader* regards code quality as a metric during grading. Thus, after RFT, the *programmer* gains enhanced capabilities in producing high-quality code while adopting edge-AI-relevant algorithms. Notably, GitHub Copilot obtains higher code quality scores than other baselines, as it employs tailored LLMs and prompts for high-quality code generation, whereas other agents are typically built on vanilla LLMs. *In general, the superior performance of SE<sup>2</sup> stems from its self-exploring capabilities in adopting SOTA algorithms dedicated to various edge AI tasks.*

## 5.6.2 Case Study on Edge AI Applications

### Metrics

We repeat the experiment 100 times for each application and calculate the average values of the following metrics: 1) *Classification accuracy* (HAR): the portion of the correctly classified activities. 2) *Answer correct rate* (VLU): the portion of the correctly answered image Q&A pairs. 3) *Precision* (ODAD): the portion of the detected bounding boxes that are actually correct. 4) *Memory consumption*: the CPU RAM or GPU VRAM usage during the execution of the generated code. 5) *Inference time*: the time used for processing a single multimodal data batch on the target device. 6) *Execution success rate*: the portion of the programs that can be successfully executed. 7) *Wall-clock generation time*: the total time elapsed from the moment the user proposes her requirement to the synthesis of the final program.

### Evaluation on HAR (Agentic workflow in § 9.3.2)

We design a user requirement (§ 9.2.1) specifying that the multimodal HAR model should be deployed on a Raspberry Pi (8GB RAM, no onboard GPU). As shown in

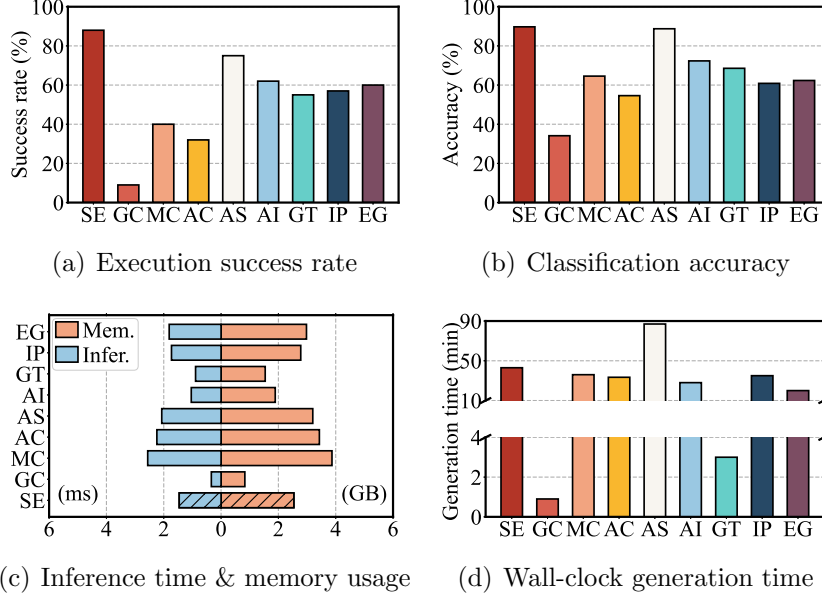


Figure 5.8: Performance comparison on HAR (SE –  $SE^2$ , GC – GitHub Copilot, MC – MapCoder, AC – AgentCoder, AS – AgentSquare, AI – AutoIOT, GT – GPIoT, IP – IoTPilot, EG – EmbedGenius).

Fig. 5.8,  $SE^2$  outperforms baselines in terms of execution success rate (39%  $\uparrow$ ) and classification accuracy (26.5%  $\uparrow$ ). Delving deeper, we find that  $SE^2$  first generates a script to obtain the data dimension, which is not specified in the user requirement. In contrast, the baselines fail as their fixed workflows often get stuck in their ways in unknown scenarios, such as unspecified data dimensions. *We note that  $SE^2$  learns to fetch such important yet missing information via self-exploration, rather than relying on explicit workflow engineering by developers.* Furthering this, by decomposing the user requirement into multiple subtasks,  $SE^2$  can independently generate, test, and refine code snippets for each, ultimately achieving a higher accuracy.

Notably, most of the programs synthesized by GitHub Copilot cannot be executed, whereas executable programs achieve only 34% accuracy on average. The reason is that GitHub Copilot is designed for general-purpose coding and lacks specialized designs. Thus, it cannot effectively produce code tailored to edge AI. This

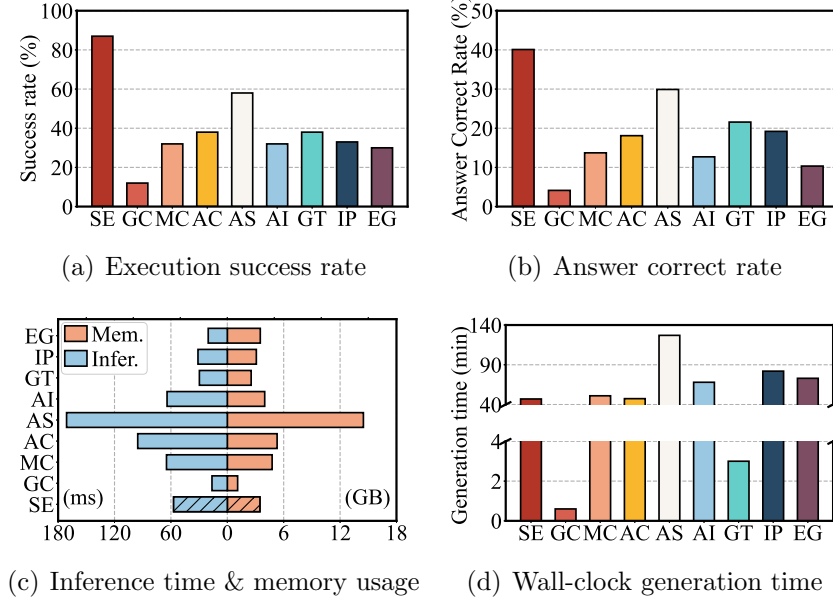


Figure 5.9: Performance comparison on VLU.

phenomenon is consistently observed across different developing tasks and is exacerbated as the edge AI task becomes more complex (Fig. 5.9 & Fig. 5.10).

Furthermore, AI models created by  $\text{SE}^2$  consume moderate memory with acceptable inference latency (Fig. 5.8(c)). Though AutoIOT and GPIoT adopt thorough model optimization methods (*e.g.*, quantization [202]) to reduce memory cost to 2GB, the resulting accuracy is much lower than  $\text{SE}^2$ . In addition, IoTPilot and EmbedGenius sometimes achieve lower performance, likely because they are more tailored to embedded tasks (*e.g.*, reading sensor data from Arduino).

Moreover, we observe that the performance of  $\text{SE}^2$ -generated programs is more stable than the baselines, achieving 2.3%, 3.5%, 0.4 ms, 0.03 GB, and 5 minutes in terms of execute success rate, classification accuracy, inference time, memory usage, and wall-clock generation time, respectively. Similar phenomena can also be observed in other AIoT applications, demonstrating that  $\text{SE}^2$  can not only provide performant programs via self-exploration and self-evolution, but also achieve lower performance fluctuations for enhanced user experience.

Last but not least,  $\text{SE}^2$  only requires around 43 minutes for program synthesis, slightly surpassing other agents that rely on fixed workflows. While GitHub Copilot and GPiOT only need a few minutes, as they involve just a few rounds of LLM calls, most synthesized programs are unexecutable. AgentSquare achieves a 13% lower success rate and takes a much longer time (90 minutes) to generate executable code, as it performs an exhaustive search over large module spaces.

### Evaluation on VLU (§ 9.3.3)

We request (§ 9.2.2) to deploy the VLU model on Jetson Nano. Unlike the x86-based PC where  $\text{SE}^2$  operates, Jetson Nano uses ARM CPUs with a distinct runtime environment for model deployment. Thus, agents must realize this case and adopt an appropriate inference engine to ensure compatibility with Jetson Nano.

As shown in Fig. 5.9,  $\text{SE}^2$  outperforms baselines in terms of both execution success rate (52.8%  $\uparrow$ ) and answer correct rate (23.9%  $\uparrow$ ). Notably, the success rate of baselines drops dramatically compared to HAR. Delving deeper into their intermediate outputs, we find that they all ignore runtime differences and repeatedly attempt to deploy x86-based models on ARM-based Jetson Nano, as their fixed workflows do not consider runtime heterogeneity. In contrast,  $\text{SE}^2$  realizes this issue by proactively analyzing the runtime semantics obtained from the probe program. Then,  $\text{SE}^2$  creates a few more agentic steps to convert the model into a platform-agnostic ONNX format [243] and use TensorRT [45] compatible with Jetson Nano for model inference (§ 9.3.3). Besides, AgentSquare needs substantial time for workflow optimization (Fig. 5.9(d)). The accumulated context resulting from it causes the agent to forget the device’s memory constraints (4GB VRAM), producing models consuming over 10GB of VRAM (Fig. 5.9(c)).

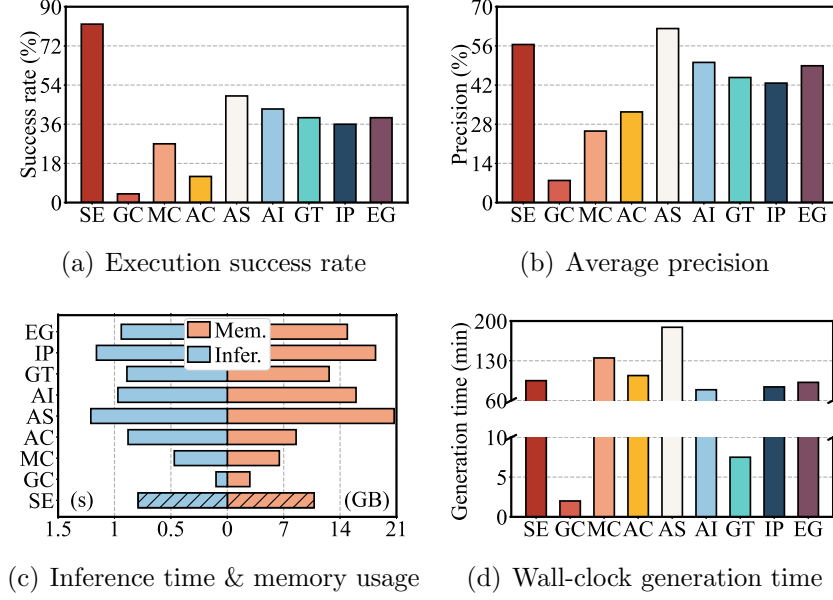


Figure 5.10: Performance comparison on ODAD.

### Evaluation on ODAD (§ 9.3.4)

For ODAD, we design a user requirement (§ 9.2.3) and request deployment on an edge laptop with 16GB VRAM. This task is highly complex as it involves four modalities with high-dimensional data. Besides, we randomly remove some modalities to simulate the modality missing issue encountered in real-world autonomous driving scenarios. Agents are expected to take this unforeseen case into account and build elastic AI models capable of processing input data of varying sizes.

As shown in Fig. 5.10,  $SE^2$  significantly outperforms baselines with an even 50% higher success rate. After analyzing its constructed workflows (§ 9.3.4), we find that though  $SE^2$  initially creates a model that fails to support elastic inputs, it discovers this flaw during testing.  $SE^2$  then modifies the model structure by agilely adjusting its workflow to cope with varying input data sizes. *Such developing pipelines closely resemble humans, exhibiting remarkable adaptability and consciousness of  $SE^2$  in handling unpredictable cases.* In contrast, the baselines repeatedly refine the programs

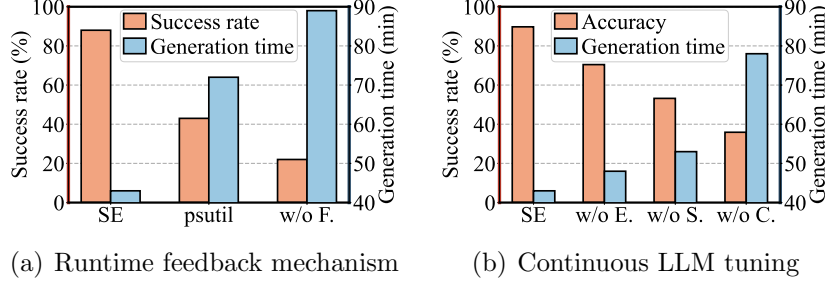


Figure 5.11: Ablation study (F. – feedback mechanism, E. – ensemble mutual-actor-critic strategy, S. – self-reflective grader, C. – continuous LLM tuning).

by randomly adopting other data processing methods or models without being aware of the modality missing issue. Worse still, they spend more time on debugging and produce high-memory-cost models that fail to meet the resource constraints.

### 5.6.3 Ablation Study

To separately test the impact of different modules, we conduct an ablation study on multimodal HAR. Similar phenomena can be observed in other applications, as showcased in § 9.6. Note that the superior performance of  $\text{SE}^2$  compared with the single-agent setting in § 5.2.2 has demonstrated the importance of our *coordinator-actor agentic architecture*.

**Mobile Runtime Feedback Mechanism.** We consider three ablation cases:  $\text{SE}^2$ ,  $\text{SE}^2$  without the probe but replaced by a widely adopted runtime monitor psutil [116], and  $\text{SE}^2$  without the entire feedback mechanism. Fig. 5.11(a) reveals that without our feedback mechanism,  $\text{SE}^2$  experiences a significant 66% drop in execution success rate and requires twice as long to synthesize an executable program. This is because, without the runtime semantics collected by the probe, the *coordinator* cannot provide useful feedback to atomic agents, thus often fails to identify whether the programs are well aligned with the target device’s runtime environments. In addition, though psutil can provide comprehensive runtime information,  $\text{SE}^2$  often struggles

Table 5.1: Participant information of user study (N=20)

| Category         | Background Information                                          |
|------------------|-----------------------------------------------------------------|
| <b>Gender</b>    | Female (45%), Male (40%), Prefer not to say (15%)               |
| <b>Age</b>       | Under 18 (5%), 18-30 (75%), 30-39 (10%), 40 and older (10%)     |
| <b>Education</b> | Bachelor (5%), Master (20%), Doctoral (70%), Others (5%)        |
| <b>English</b>   | Fluent (10%), Advanced (65%), Intermediate (20%), Beginner (5%) |

to extract content relevant to current coding subtasks, leading to a 1.5x increase in synthesis time. *These results highlight the importance of our feedback mechanism in timely providing runtime semantics grounded in mobile devices to enhance edge AI development.*

**Continuous LLM Tuning.** We test four ablation cases:  $SE^2$ ,  $SE^2$  without the *ensemble mutual-actor-critic strategy*,  $SE^2$  without the *self-reflective grader*, and  $SE^2$  without the entire tuning module. Fig. 5.11(b) shows that all modules are crucial for retaining a high execution success rate and reducing code generation time. Without the mutual-actor-critic strategy, the *coordinator* and atomic agents cannot be mutually updated, resulting in obsolete reasoning abilities when facing new edge AI knowledge and tasks. Without the self-reflective grader, the *programmer* cannot reflect on its generated algorithms and continuously evolve to produce more edge-AI-relevant programs. *These results show the effectiveness of our continuous LLM tuning paradigm in progressively enhancing agents’ reasoning abilities for edge AI tasks.*

#### 5.6.4 User Study for Generalizability Test

To assess the utility and overall user satisfaction of  $SE^2$ , we conduct a user study<sup>‡</sup> involving 10 expert users and 10 non-expert users. Expert users have rich experience in sensor data processing and AI technologies, while non-expert users only possess basic knowledge of edge AI applications. *Importantly, to test the **generalizability***

<sup>‡</sup>The evaluation was approved by our institution’s ethics review board (IRB).

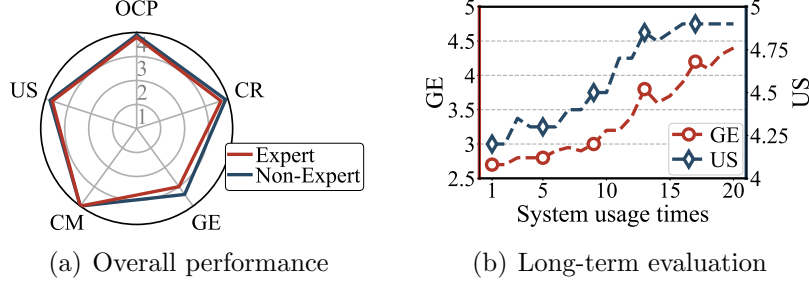


Figure 5.12: Evaluation results of the user study.

of  $SE^2$ , we did not limit the tasks. Instead, users are encouraged to freely express their requirements for developing various edge AI applications. Table 5.1 lists the detailed background information of the participants.

## Metrics

1) *Overall code performance (OCP)* evaluates the overall performance (*e.g.*, task accuracy, resource consumption) of the synthesized programs; 2) *Code readability (CR)* measures the clarity and structure of the synthesized programs; 3) *Generation efficiency (GE)* accesses how efficiently  $SE^2$  operates in terms of speed and resource usage to produce the final code; 4) *Code modularity (CM)* judges whether the code is properly modularized for easy reuse and extension; 5) *User satisfaction (US)* captures users' overall experience with  $SE^2$ . We ask users to rate these metrics on a scale from 1 (not at all) to 5 (completely).

## Overall Performance

Users' improvisational requirements span a range of edge AI topics, including mobile sensing (40%), actuator control (40%), and personalized services (20%). Experts often challenge  $SE^2$  with more complex tasks (*e.g.*, multimodal emotion analysis) while non-experts tend to propose simple tasks (*e.g.*, IMU-based HAR). In Fig. 5.12(a), both experts and non-experts provide high scores (above 4.7) in terms of OCP, CR,

CM, and US. Notably, nearly all of them award full marks for code modularity. After examining the synthesized programs, we attribute this to our design of atomic operations and hierarchical agentic architecture, which enables different agents to exploit their own reasoning abilities to progressively explore and generate modular code to implement various functionalities. *This process is akin to how humans decompose a task into multiple modules and generate modular code for each.* Besides, experts tend to provide lower GE scores than non-experts, as they are knowledgeable in edge AI and have higher expectations for system performance. Though the GE score is relatively lower, it increases as system utilization grows (§ 5.6.4), benefiting from the *continuous tuning paradigm*. *These results exhibit the **generalizability** of  $SE^2$  in exploring distinct workflows for diverse edge AI tasks with enhanced user experience.*

### Long-Term Evaluation

We further ask expert users to participate in a long-term evaluation to test the effectiveness of our self-evolving paradigm. Fig. 5.12(b) shows that as user engagement increases over time, both GE and US rise with a similar growth trend. As system usage time increases, the mutual scores graded by the *coordinator* and atomic agents gradually accumulate and are used in mutual-critique for reasoning enhancement. Moreover, the *retriever* progressively stores newly fetched knowledge in the local database for timely reference. As such, when new similar tasks are assigned,  $SE^2$  can activate fast thinking to make decisions rapidly without re-exploring algorithms, resulting in shorter program synthesis times. Notably, though  $SE^2$  initially tends to adopt wrong algorithms due to hallucinations, they are corrected by the feedback mechanism, and the likelihood decreases as RFT progresses. *These results demonstrate the outstanding self-evolving capabilities of  $SE^2$  in handling diverse user requirements and emerging edge AI knowledge.*

## 5.7 Discussion

**Vibe Coding vs. SE<sup>2</sup>.** The emerging concept of vibe coding refers to using LLMs to generate code snippets or individual functions purely from user instructions, without execution and verification [244]. Users must manually design multiple prompts and verify the correctness of each generated code snippet individually. In contrast, SE<sup>2</sup> provides an end-to-end solution that automatically translates user requirements into executable, runtime-compatible, and performant on-device programs. By fully exploiting the reasoning abilities of LLMs, SE<sup>2</sup> autonomously determines and adaptively adjusts appropriate agentic workflows for robust edge AI development.

**Error Handling in Agentic Systems.** In multi-agent systems, a common challenge is the generation of incorrect intermediate results during multi-step reasoning. These errors can propagate and lead to incorrect final outputs, significantly degrading agents' reliability. Therefore, we further design an *agentic recovery mechanism* (§ 9.5) to detect deviations from expected behavior and rollback to re-plan. Moreover, SE<sup>2</sup> orthogonally incorporates Reflexion [245] that enables agents to reflect on failed attempts for iterative refinement.

## 5.8 Related Work

**Agents & Code Generation.** By equipping LLMs with various external tools, AI agents can automate complex tasks, such as phone manipulation [63] and IoT data interpretation [71]. Moreover, many works build automated agents to synthesize various IoT programs. For example, AutoIOT [4] features an IoT knowledge retrieval module and iterative refinement for code generation. EmbedGenius [7] introduces a hardware-in-the-loop paradigm that enables agents to flash generated code onto embedded devices. However, developers must meticulously design and optimize the agentic workflows to ensure that agents can follow explicitly predefined guidance,

limiting their flexibility and adaptability. Instead,  $\text{SE}^2$  exploits LLMs’ own reasoning abilities to autonomously construct workflows for edge AI development.

**Agentic Workflow Optimization.** Some recent studies [61, 62] focus on automated optimization of predefined workflows to reduce human effort. For instance, ADAS [61] and AgentSquare [3] regard predefined workflows as combinations of various agentic modules (*e.g.*, Chain-of-Thought [87] and Tree-of-Thought [246] for reasoning). Then, they perform an exhaustive search within constrained spaces to identify the optimal combination that maximizes performance. Yet, they still rely on fixed, predefined workflows bound by developers’ experiences. Instead, drawing strengths from these arts,  $\text{SE}^2$  shifts the burden of workflow design from developers to LLM agents, enabling more flexible and adaptive edge AI development. Notably,  $\text{SE}^2$  can also be orthogonally integrated with these arts for further optimization.

## 5.9 Chapter Summary

We present  $\text{SE}^2$ , a self-exploring and self-evolving paradigm for adaptive edge AI development. By fully exploiting the reasoning abilities of LLMs,  $\text{SE}^2$  agents can autonomously explore distinct appropriate workflows for various edge AI tasks. Meanwhile, our *continuous LLM tuning* paradigm progressively strengthens agents to tackle evolving edge AI knowledge and environments. Extensive evaluations demonstrate the *effectiveness*, *generalizability*, *flexibility*, and *adaptability* of  $\text{SE}^2$  in handling diverse and evolving edge AI tasks. Looking ahead, we envision  $\text{SE}^2$  as a stepping stone toward more autonomous agentic systems capable of adapting to dynamic environments as well as evolving through lifelong learning and reflection.



# Chapter 6

## Automating AIoT Model Deployment on Heterogeneous Mobile Devices

### 6.1 Introduction

Embedded AI aims to support intelligent computation directly on resource-constrained devices and enable real-time sensing and decision-making while preserving user privacy on local devices [69, 247, 248, 249, 250, 251, 8]. However, delivering such intelligence is still far from plug-and-play, since deploying pretrained models across a range of heterogeneous embedded platforms often encounters severe compatibility and performance penalties.

As illustrated in the typical deployment workflow (Fig. 6.1), developers first download a pretrained AI model from open-source platforms such as Hugging Face [252] or GitHub [253]. Then, they need to select and configure an inference engine (serving as middleware between the model and the target embedded device) for on-device model execution. In practice, however, many pre-trained models cannot fully utilize a single inference engine.

The root cause of this model deployment issue lies in the mismatch between pre-trained AI models and inference engines. Specifically, an AI model is composed of

multiple operators (*e.g.*, **Add**, **Conv**) that perform various computational tasks. Inference engines, however, are developed and optimized for distinct hardware architectures and runtime environments. For instance, TensorRT [45] is suited for NVIDIA Jetson Nano, while OpenVINO [46] targets Intel GPU-based platforms. Hence, an inference engine is highly optimized for a small subset of operators that best match its runtime and hardware backends, leaving the rest either unsupported or only partially supported at best. This *model-engine mismatch* issue leads to sub-optimal performance or even failures in model inference.

Our preliminary study across 150 open-source AI models indicates that 38 models (*i.e.*, 25.3%) fail to execute on a given embedded device due to *model-engine mismatch* (detailed in § 6.2.2). Further analysis reveals that this model-engine mismatch manifests in three forms with performance implications: 1) Certain operators are missing in some engines (*e.g.*, TensorRT does not support **Mod**); 2) Many pretrained models customize operators, which are not supported by any inference engines; 3) Engine version inconsistency (*e.g.*, adding new or deprecating old operators) further exacerbates the mismatch. Facing such mismatch issues, developers may resort to slower CPU runtimes (rather than GPUs or other onboard accelerators), which inevitably results in degraded performance in terms of inference latency. Another workaround requires model deployers (typically users or application developers) to painstakingly register all missing operators into the target inference engine. While feasible, this approach requires in-depth knowledge of both model internals and engine APIs, as well as operator customization and augmentation.

In this thesis, we propose **AutoDeploy**, an end-to-end system that automates model deployment on heterogeneous embedded devices with the objective of addressing the *model-engine mismatch* issue. As illustrated in Fig. 6.1, instead of relying on a single inference engine, **AutoDeploy** strategically partitions a pretrained model into several segments, executes each segment on its most suitable engine, and directs

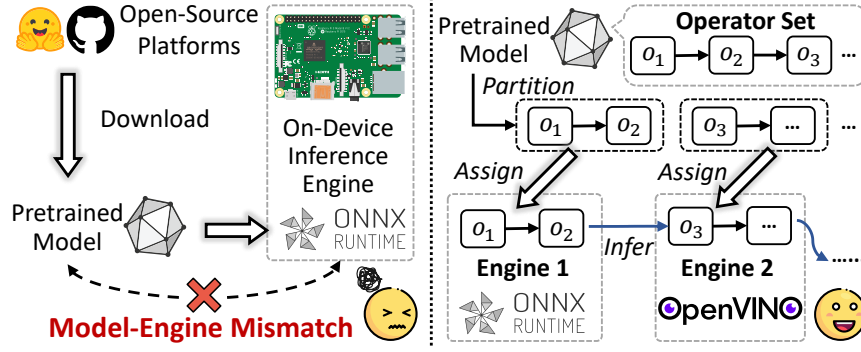


Figure 6.1: Left: a typical workflow of embedded AI model deployment and the *model-engine mismatch* issue. Right: the high-level idea of AutoDeploy.

intermediate results across engines for seamless on-device inference. As such, operators in AI models can be collectively supported by multiple inference engines for efficient on-device execution. Although this basic idea may appear straightforward, we face significant technical challenges in practice:

Challenge-1: It is challenging to design a one-size-fits-all solution to check the compatibility between model operators and inference engines. For the same operator, both AI models and inference engines adopt distinct naming conventions. This difference in naming can lead to challenges when analyzing models across frameworks or deploying them on heterogeneous devices. Worse still, the set of engine-supported operators—including newly introduced or recently deprecated ones—varies across engine versions. As a result, static, rule-based compatibility checks may fail to determine whether the operators in a pretrained model are supported by a specific engine. To address this challenge, we develop a new *agent-based operator analyzer* that employs AI agents to automatically retrieve up-to-date operator knowledge from official documentation and analyze compatibility. The underlying rationale is twofold: 1) AI agents can effectively examine and reason over the source code of pretrained models and inference engines to judge if their operators implement equivalent computational functionalities despite the differences in specific operator names; 2) Guided by

prompts and external tools, agents can automate the process of compatibility check with minimal user intervention [254]. In addition, they can autonomously detect and recover from mistakes through safeguard mechanisms to ensure high robustness and accuracy [245].

Challenge-2: The appropriate partition granularity is hard to determine. Different engines adopt diverse yet incompatible internal optimization techniques. For example, some optimization strategies merge multiple operators (*e.g.*, convolution and addition) into more efficient composite units, while others eliminate redundant operators based on predefined operator patterns and their interplays. Naively partitioning pretrained models solely based on operator-engine compatibility can break these optimizable patterns and ultimately undermine inference efficiency. To tackle this practical issue, we design a *structure-aware pattern detector* that analyzes the source code of various engines to dynamically detect optimizable patterns. The detected patterns are then explicitly considered as optimization constraints to preserve the optimizable structures, facilitating the corresponding engine-level optimization.

Challenge-3: Randomly assigning model partitions to compatible inference engines often leads to suboptimal performance, highlighting the need for principled optimization strategies. Due to the heterogeneity of embedded runtime environments, different engines take varying computation times for the same operations (§ 6.4.3). For instance, ONNX Runtime [255] executes `Conv` faster but takes a longer time for `LSTM` compared to OpenVINO [46]. Overlooking such performance discrepancies affects overall inference performance. Worse still, frequent engine switching introduces extra overhead due to context transitions. To overcome these limitations, we propose a new *greedy cross-engine inference algorithm* that jointly accounts for operator-engine compatibility, engine-specific operator execution time, and operator-switching cost during model partitioning and cross-engine execution.

We implement and comprehensively evaluate **AutoDeploy** on heterogeneous em-

bedded devices with 150 open-source AI models across multiple representative tasks (*image recognition, speech sentiment analysis, and vision-language understanding*) that can support innovative downstream applications. We compare **AutoDeploy** with five state-of-the-art (SOTA) inference engines and human developers by recruiting fifteen users with different expertise and proficiency in embedded AI development. Extensive experiments show that **AutoDeploy** significantly improves the deployment success rate to nearly 100% after effectively resolving the *model-engine mismatch* problem. **AutoDeploy** can also reduce model deployment time (*i.e.*, from user requirement specification to successful model deployment) by approximately 79.3% compared with even experienced expert developers, who are allowed to freely employ techniques such as manual patching, model adaptation, and even AI agents. Moreover, **AutoDeploy** can reduce model inference latency by approximately 26.6% on average compared with SOTA inference engines, when we manually register with all necessary operators. Last but not least, by allowing users to freely specify pretrained AI models for deployment, our user studies demonstrate *high generalizability* of **AutoDeploy** in deploying diverse open-source models on heterogeneous embedded devices.

In summary, we make the following contributions:

- **AutoDeploy** presents the first end-to-end system that mitigates the *model-engine mismatch* issue by fully automating model deployment on heterogeneous embedded devices with minimal user intervention and reduced inference latency.
- **AutoDeploy** features three new technical modules to enhance efficiency and robustness: an *agent-based operator analyzer* for up-to-date operator analysis, a *structure-aware pattern detector* for preserving optimizable operator patterns, and a *greedy cross-engine inference algorithm* for overall latency optimization.

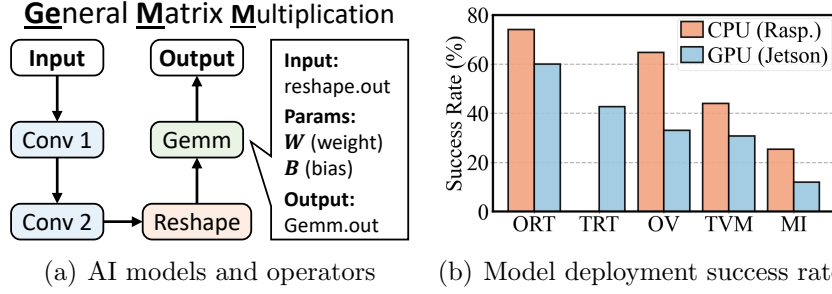


Figure 6.2: (a) The model structure. (b) The model deployment success rate across various types of models and inference engines (TensorRT does not support CPU mode). (ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO, MI – MILR)

- Comprehensive evaluations validate the effectiveness and efficiency of **AutoDeploy**, demonstrating its ability to significantly eliminate *model-engine mismatch* and reduce inference latency when deploying AI models on diverse embedded devices.

## 6.2 Background & Motivation

### 6.2.1 Embedded Model Deployment

To understand how AI models are deployed and executed on embedded devices, we construct a model comprising two convolutional layers and one fully connected layer for image classification tasks.

As illustrated in Fig. 6.2(a), the model is represented as a computation graph\*, where nodes correspond to operators performing specific computational functions, and edges denote data dependencies that define the flow of data between operators. For instance, the General Matrix Multiplication (**Gemm**) operator receives the output of the preceding operator, performs matrix multiplication, and produces a new output. During inference, an input image traverses this graph through a series

\*Computation graphs of existing open-source AI models are commonly serialized into various model formats such as PyTorch [256], TensorFlow [47], ONNX [243], *etc.*

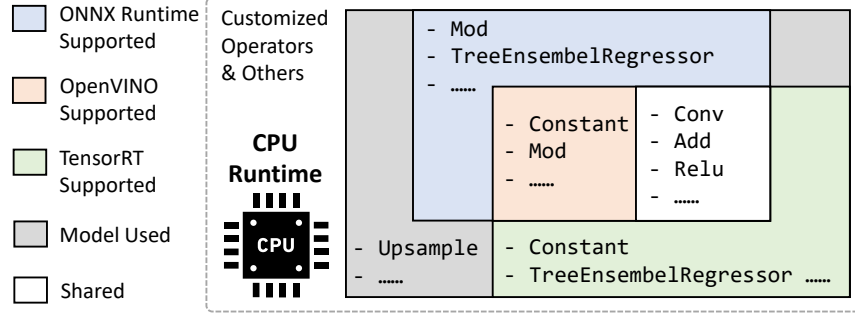


Figure 6.3: The mismatch between AI models and engines.

of sequential or parallel operators, ultimately generating the final output, *i.e.*, the predicted image class.

To deploy the model on a target embedded device, developers typically select a suitable inference engine (*e.g.*, ONNX Runtime [255]) to execute the model. Inference engines parse the computation graph, assign optimized kernel implementations for each operator, schedule their execution efficiently, and manage memory and data flow at runtime. As such, serving as model accelerators, inference engines provide hardware-aware, runtime-optimized execution that reduces latency, memory usage, and energy consumption.

## 6.2.2 Model-Engine Mismatch

In practice, however, deploying AI models on heterogeneous embedded devices often falls short of expectations. We observe that many models cannot be successfully executed by a single inference engine. The culprit is ***model-engine mismatch***: the operators used in an AI model may not be fully supported by an engine. For example, the `Mod` operator is not supported by TensorRT (Fig. 6.3). To investigate the *model-engine mismatch* issue, we conduct a preliminary study analyzing various AI models and inference engines.

**Model & Inference Engine Selection.** We first download 150 open-source models from Hugging Face, covering distinct architectures, inference tasks, and model

Table 6.1: Detailed information of the 150 downloaded models.

| Types                       | Statistics                                                                              |
|-----------------------------|-----------------------------------------------------------------------------------------|
| <b>Model Architecture</b>   | Feedforward Networks (6.7%), CNN (13.3%), RNN (6.7%), Transformer (20%), Hybrid (53.3%) |
| <b>Inference Task</b>       | Natural Language Processing (30%), Computer Vision (30%), Multimodal Application (40%)  |
| <b>Model Size</b>           | 0~1B (66.7%), 1B~6B (16.7%), 6B~12B (10%), >12B (6.7%)                                  |
| <b>Customized Operators</b> | Contain (21.3%), Does not contain (78.7%)                                               |

sizes. Detailed information on these models is summarized in Table 6.1. Furthermore, we select five SOTA inference engines: ONNX Runtime [255], TensorRT [45], OpenVINO [46], TVM [257], and MLIR [258].

**Model Deployment & Result Analysis.** For fair comparison and illustration purposes, all models are deployed on two embedded devices with distinct runtime environments: a Raspberry Pi (CPU runtime) and a Jetson Nano board (GPU runtime). We utilize the five inference engines for model execution and record their deployment success rates. As shown in Fig. 6.2(b), all engines achieve deployment success rates below 80% in both CPU and GPU inference modes, indicating that at least 30 out of the 150 models fail to execute. Moreover, the GPU mode exhibits a lower success rate than the CPU mode. Further examination of the failed cases reveals the *model-engine mismatch* problem in three critical aspects:

- **Unsupported Operators.** 38% of the failed models contain some operators that cannot be computed by a single engine. For instance, the Reformer model [259] adopts a `Mod` operator, which is not supported by TensorRT but supported by ONNX Runtime.
- **Developer-Customized Operators.** 62% of the failed models (*e.g.*, Deformable DETR [260], BigVGAN [261]) employ several developer-customized operators, which are inherently incompatible with all the selected inference

engines.

- **Harsh Software Dependencies.** Worse still, nearly 80% of the above cases are caused by heterogeneous runtime environments. Specifically, different devices support distinct versions of inference engines, which may introduce new operators or deprecate existing ones [262]. Such heterogeneity further aggravates the *model-engine mismatch* issue across diverse devices and engines.

To overcome these difficulties, developers have to manually implement additional code to explicitly register unsupported or customized operators into specific inference engines. Otherwise, they have to resort to less efficient fallback environments (*e.g.*, CPU) for model execution. These processes are highly labor-intensive and error-prone, demanding specialized domain expertise and substantial human effort from developers.

### 6.2.3 AutoDeploy & Motivation

The findings from our preliminary study motivate the development of a more efficient approach for embedded AI model deployment. To this end, we propose **AutoDeploy** to mitigate the *model-engine mismatch* issue and lower the expertise barrier to deploying AI models on heterogeneous embedded devices. Specifically, given a pre-trained model, **AutoDeploy** partitions it into multiple segments, each can be executed by a compatible inference engine. As such, the pretrained model’s operators can be collectively supported by the union of multiple inference engines. By organically scheduling the execution of model partitions and their corresponding inference engines, **AutoDeploy** enables effective and efficient deployment of various pretrained AI models on heterogeneous embedded devices.

To realize this vision, we address three unique technical challenges (§ 6.1) and design three novel solutions accordingly. First, we develop an *agent-based operator*

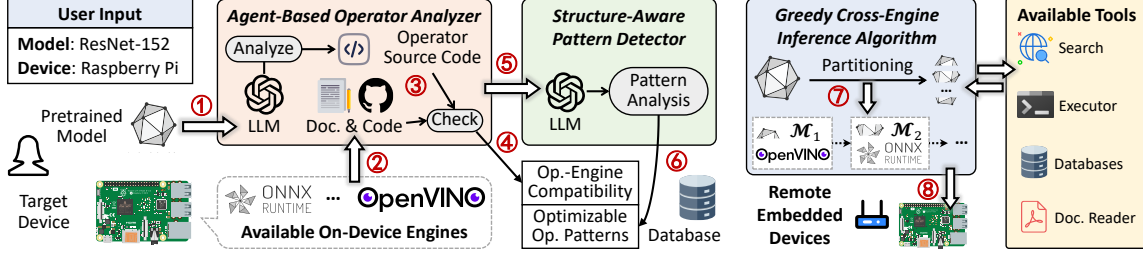


Figure 6.4: The system overview and workflow of AutoDeploy with three novel technical modules.

*analyzer* to deal with highly diverse operator naming conventions. Second, to preserve engine-specific operator optimization strategies, we introduce a *structure-aware pattern detector* that identifies and records optimizable operator patterns, serving as constraints during model partitioning. Finally, we propose a *greedy cross-engine inference algorithm* to orchestrate model partitions and inference engines for on-device model execution, meanwhile minimizing overall inference latency.

## 6.3 System Overview

### 6.3.1 AutoDeploy Workflow

Fig. 6.4 shows the overall workflow of AutoDeploy, comprising three technical modules that automate model deployment on various embedded devices. AutoDeploy requires the following prerequisites:

- AutoDeploy runs on an edge server and accesses target embedded devices via a local network for remote control.
- The target device is equipped with multiple inference engines.
- Model inference and evaluation are performed entirely on target devices to preserve data privacy [263].
- We focus on model deployment and assume user-provided models are affordable

by resource-constrained target devices.

- We provide the following tools for the AI agent in **AutoDeploy**: a web search engine, a vector database, a Python executor within a sandbox, and a document loader.

**Workflow.** From the user’s perspective, she only needs to specify a target embedded device and provide a pretrained model that meets the device’s resource constraints (❶). The *agent-based operator analyzer* (§ 6.4.1) first retrieves the official documentation and source of the on-device inference engines (❷). The analyzer then checks the compatibility between each operator in the pretrained model and the engine-supported operators (❸). Meanwhile, the operator-engine compatibility information is stored in a continuously updated database (❹). Subsequently, for each inference engine, the *structure-aware pattern detector* (§ 6.4.2) interprets its documentation to identify optimizable operator patterns (❺) and records the corresponding set of optimizable patterns residing in the pretrained model (❻). Finally, the *greedy cross-engine inference algorithm* (§ 6.4.3) partitions the model into multiple segments (❼) by greedily considering operator-engine compatibility, optimizable patterns, and engine-switching overhead. The model partitions are then alternately executed across different inference engines (❽).

**Remarks.** The operator analyzer and pattern detector employ an AI agent to dynamically retrieve up-to-date online information and maintain a continuously updated database, thereby significantly reducing human effort and expertise requirements. Note that the knowledge retrieval and database updating can be offline processed.

### 6.3.2 Problem Formulation

We formulate the problem of model partitioning and execution via multiple inference engines for various embedded AI applications.

**Definitions.** An AI model  $\mathcal{M}$  is represented as a directed acyclic graph (Fig. 6.2(a)),  $G = (V, E)$ , containing a set of vertices  $V$  and edges  $E$ . Each vertex corresponds to an operator  $o_i$  that performs a specific computation task, such as addition or matrix multiplication. Each edge denotes the data dependency and flow between operators, defining how outputs of one node serve as inputs to another during model execution. As such, the model can be expressed as:

$$\mathcal{M} \sim G = (V, E), \quad V = \{o_1, o_2, \dots, o_n\}, \quad E^{n \times n} \subseteq V \times V \quad (6.1)$$

where  $E^{n \times n}$  can be regarded as a binary-valued adjacency matrix, where  $E(i, j) = 1$  indicates that  $o_j$  takes the output of  $o_i$  as its input.

**Problem Statement.** Given a user-specified pretrained model,  $\mathcal{M}$ , and a list of on-device inference engines,  $\mathcal{E} = \{e_1, e_2, \dots, e_m\}$ , running on the target embedded device, **AutoDeploy** aims to partition the model into multiple segments,  $\{\mathcal{M}_1, \mathcal{M}_2, \dots\}$ , each can be successfully executed by an inference engine,  $e_j$ . Our goal is to find such a partition method,  $P$ , meanwhile achieving minimal model inference latency. The problem can be expressed as:

$$\begin{aligned} P = & \text{minimize} \quad \text{inference latency} \\ & \mathcal{M} \xrightarrow{P} \{\mathcal{M}_1, \mathcal{M}_2, \dots\}, \quad \mathcal{M}_j \sim G_j = (V_j, E_j) \\ \text{s.t.} \quad & V = \bigcup V_j, \quad E = \bigcup E_j \\ & \exists e_j \in \mathcal{E} : V_j = \{o_1^j, o_2^j, \dots\} \subseteq \mathcal{O}_{e_j} \\ & \mathcal{O}_{e_j} = \{o_1^{e_j}, o_2^{e_j}, \dots\} \end{aligned} \quad (6.2)$$

where all operators  $V_j$  of the  $j$ -th model partition  $\mathcal{M}_j$  are fully compatible with an on-device inference engine,  $e_j$ , whose supported operators are  $\mathcal{O}_{e_j}$ . As such, the on-device model inference process can be viewed as alternately using these engines

to deliver intermediate outputs and execute each model partition sequentially:

$$\mathbf{Output} = e_1(\mathcal{M}_1(\mathbf{Input})) \rightarrow e_2(\mathcal{M}_2(\cdot)) \rightarrow \dots \quad (6.3)$$

## 6.4 System Design

### 6.4.1 Agent-Based Operator Analyzer

Based on the user-provided pretrained model, this module checks the compatibility between each operator in the model and the inference engines running on the target embedded device.

**Challenge.** A straightforward approach is to construct a compatibility mapping table that associates each operator in the pretrained model with its supported engines. However, delving deeper into various model operators and engines, we find that the mapping space is extremely large, making it impractical to rely on manual or rule-based scripts for operator analysis:

- *Diverse naming conventions.* The same operator can have different names across both AI models and inference engines. For example, a 2D convolution operator is named `Conv2d` in PyTorch models, whereas OpenVINO refers to it as `aten::conv2d`. Moreover, certain inference engines even define multiple distinct operators to handle different input conditions (*e.g.*, `Conv` and `ConvInteger` in ONNX Runtime [255]).
- *Varying engine versions.* Worse still, due to heterogeneous software dependencies and runtime environments, embedded devices often host inference engines of different versions, which further exacerbate the heterogeneity of their supported operators.

Therefore, instead of designing a one-size-fits-all script to judge operator-engine compatibility, we should *conduct an in-depth analysis of the implementation details (i.e.,*

*source code) of AI models and inference engines to compare their underlying execution logic and design rationale for compatibility check.*

**Solution & Rationale.** Manually analyzing the source code demands substantial domain expertise and human effort, making the process labor-intensive and error-prone. To tackle this problem, we propose an *agent-based operator analyzer* that employs an AI agent for code analysis and compatibility check. The underlying rationale of this design is twofold. 1) The large language models (LLMs) inside AI agents are pretrained on extensive domain knowledge, exhibiting exceptional code interpretation capabilities [7, 5, 264]. 2) Equipped with various external tools (*e.g.*, web search engines and databases), AI agents can autonomously retrieve and store up-to-date operator information for continuous updating and timely analysis. Leveraging such advantages, AI agents can effectively and efficiently check the compatibility between operators and inference engines without developer intervention.

Specifically, the operator analyzer consists of two stages. 1) *Information Retrieval with Maintenance*. We first prompt the agent to retrieve the source code of various model operators and inference engines from their official online documentation and GitHub repositories. The retrieved knowledge is further stored in a local database that will be dynamically maintained on a regular basis. 2) *Compatibility Check via Code Analysis*. Based on the knowledge database, the next step is to check the compatibility between each operator in the pretrained model and the engine-supported operators by analyzing their source code. If an engine-supported operator implements the same functionality as the operator in the pretrained model, it is then marked as compatible with that engine.

**Information Retrieval with Maintenance.** For each on-device inference engine, we first prompt the agent to retrieve its source code and identify a list of supported operators. For each operator, **AutoDeploy** then requests the agent to extract the description of the operator from the inference’s official documentation. Furthermore,

the agent is instructed (P.1) to locate and extract the corresponding implementation code segment. Finally, we instruct the agent to organize all collected information into four structured operator attributes, including the operator name, functionality description, I/O specifications, and implementation code.

**P. 7: Operator Implementation Code Extraction (simplified)**

You are a **code analyzer** specialized in understanding inference engines (*e.g.*, TensorRT, ONNX Runtime). Your task is to locate and extract the implementation code segment corresponding to a given operator.

**Objectives:**

1. Search the repository and locate the exact file(s) and function(s) implementing the operator.
2. Extract the relevant code segment that includes:
  - The operator class or kernel definition.
  - Its core compute function or forward implementation.
  - Input/output data types, dimensions, and semantics of the operator.
  - Key attributes or registration macros that link to the operator name.
3. Summarize, format, and output only the extracted code segment.

**Inference Engine:** {...}; **Repository URL:** {...}; **Operator Name:** {...}

Mathematically, given an inference engine  $e_j$ , the extracted information of its supported operator list  $\mathcal{O}_{e_j}$  can be expressed as:

$$\mathcal{O}_{e_j} = \{o_1^j, o_2^j, \dots\}, \quad o_i^j = \langle \text{name, functionality, IO, code} \rangle \quad (6.4)$$

These operator attributes are stored in a local database  $\mathcal{D}$  for further analysis, serving as the context knowledge of operators:

$$\mathcal{D} = \{\mathcal{O}_{e_1}, \mathcal{O}_{e_2}, \dots\} \quad (6.5)$$

**Compatibility Check via Code Analysis.** To check operator-engine compatibility via code analysis, unlike existing works [265, 266] that examine syntax trees or generate unit test cases for code analysis, our prompt (P.2) considers two unique aspects:

- *Functionality.* They must implement the same computational functionality.

For instance, the **Conv2d** operator in a PyTorch model is compatible with

OpenVINO, as it supports the `aten::conv2d` operator that performs the same 2D convolution computation.

- *I/O Match.* They must have identical I/O dimensionalities and data types. For example, if a quantized model accepts integer inputs, the corresponding engine-supported operator should also support integer computation (*e.g.*, `ConvInteger` in ONNX Runtime).

#### P. 8: Compatibility Check Prompt (simplified)

You are a **code auditor** specialized in inference engines. Your goal is to determine whether an operator from a pretrained model is compatible with an engine-supported operator by analyzing their source code.

##### Objectives:

1. Understand operator functionality
  - \* Determine what computational operation they perform.
  - \* Compare their mathematical formulations, core loops, or library calls.
  - \* Confirm if both operators achieve the same computational semantics.
2. Analyze I/O specifications
  - \* Extract the list of input and output tensors.
  - \* Record each tensor's **dimensionality**, **data type**, and **role**.
  - \* Identify and note any mismatch.
3. If both the computational functionality and I/O specifications match, only output "compatible"; otherwise, only output "no".

**Pretrained model operator:** {...}; **Engine-supported operator:** {...}

If at least one engine-supported operator satisfies the above judgment rules, the operator  $o_i$  in the pretrained model is marked as compatible with the current inference engine  $e_j$ . Such an agent-based judgment process can be expressed as:

$$\begin{aligned} \mathcal{C}_i^j &= C(o_i, o_1^{e_j}) \vee C(o_i, o_2^{e_j}) \vee \dots, \quad \forall o_i^{e_j} \in \mathcal{O}_{e_j} \\ C(o_i, o_1^{e_j}) &= \mathcal{L}(o_i, \langle \text{name, functionality, IO, code} \rangle \mid \text{P.2}) \end{aligned} \quad (6.6)$$

where  $\mathcal{C}_i^j = 1$  means the agent determines that operator  $o_i$  is supported by engine  $e_j$ .  $\mathcal{L}$  is the black-box LLM function based on **P.2**. By traversing all operators in the pretrained model  $\mathcal{M}$ , we obtain a list of supported engines for each operator, expressed as:

$$\mathcal{M} \sim \{o_1, o_2, \dots\}, \quad o_i \sim [\mathcal{C}_i^1, \mathcal{C}_i^2, \dots] \quad (6.7)$$

In addition, the list of supported engines for each operator is stored in the database to enable efficient reuse and timely updates. Note that we also instruct the agent to generate programs to register developer-customized operators into on-device inference engines.

**Remarks.** AutoDeploy exploits agents to thoroughly excavate the underlying execution logic of operators via code analysis. Thus, the cumbersome process of operator analysis and compatibility check can be delegated to the automated, knowledge-armed agent without user intervention. Additionally, the agent is periodically triggered to re-retrieve up-to-date knowledge and refresh the database.

### 6.4.2 Structure-Aware Pattern Detector

With the obtained information on operator-engine compatibility, we could intuitively partition the pretrained model into several segments randomly, each fully supported by a single inference engine for execution. However, we find that this vanilla partitioning strategy often leads to increased inference latency.

**Challenge.** We deploy Deformable DETR [260] (hybrid architecture) on a Jetson Nano board (GPU runtime) and a Raspberry Pi (CPU runtime). We record the inference time of a single data batch under two settings: 1) using the vanilla partitioning method described earlier, and 2) manually registering all model operators into three SOTA inference engines (ONNX Runtime, TensorRT, and OpenVINO). As shown in Fig. 6.5(a), the vanilla method yields substantially longer model inference time than using a single inference engine, even under GPU runtime. A deeper examination of the model partitions and their corresponding inference engines reveals that the vanilla partitioning method overlooks a critical factor: *optimizable operator patterns*. Typically, inference engines employ two types of internal operator optimization strategies: merging consecutive operators and removing redundant ones based on predefined patterns. For instance, in Fig. 6.5(b), ONNX Runtime fuses the

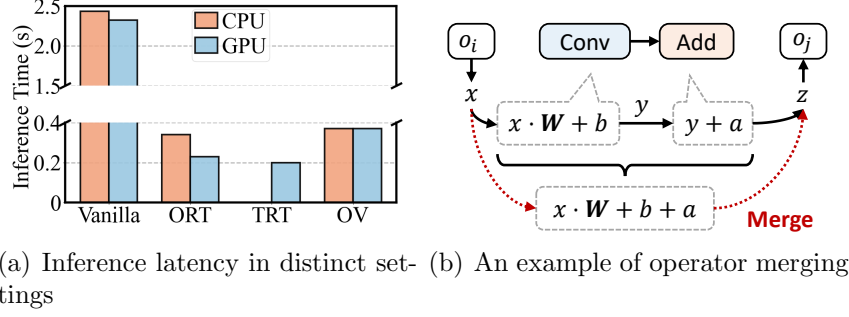


Figure 6.5: (a) The vanilla partitioning method results in significantly longer inference time (ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO); (b) An example of operator optimization: merging the **Conv** and the **Add** operator.

**Conv** and **Add** operators by folding the **Add** operator as the bias of the **Conv** operator [267]. Therefore, partitioning AI models simply based on operator-engine compatibility may damage these optimizable patterns, leading to unnecessary increases in inference time.

**Solution & Rationale.** Taking optimizable patterns into account, we propose a *structure-aware pattern detector* that operates in two stages: *pattern mining* and *model reorganization*. 1) *Pattern Mining*. The detector analyzes the source code of each on-device inference engine to extract and record its internal optimizable patterns. 2) *Model Reorganization*. We first clone the pretrained model into multiple copies, each corresponding to an on-device inference engine. The detector then reorganizes each model copy by merging consecutive operators into unified computational units or removing redundant ones based on the mined optimizable patterns with respect to each on-device engine. These patterns and the reorganized model are further used as constraints in our *greedy cross-engine inference algorithm* (§ 6.4.3). By keeping optimizable patterns intact and fully exploiting engine-specific optimization mechanisms, **AutoDeploy** enables more efficient cross-engine model execution.

**Pattern Mining.** For each on-device inference engine  $e_j$ , the agent first parses its documentation to identify a list of internal optimizable operator patterns. Next,

the agent is instructed to extract involved operators and locate the corresponding implementation code for each optimizable pattern in the list:

$$\mathcal{P}_j = \{p_1^j, p_2^j, \dots\}, \quad p_r^j = \langle \mathcal{O}_r^j, c_r^j \rangle, \quad \mathcal{O}_r^j = \{o_1^j, o_2^j, \dots\} \quad (6.8)$$

where  $\mathcal{P}_j$  is the identified pattern list of the current engine  $e_j$ ,  $\mathcal{O}_r^j$  is the optimizable operators of the  $r$ -th pattern, and  $c_r^j$  is its implementation code, which will be further used for model reorganization.

**Model Reorganization.** Next, we clone the pretrained model into multiple copies, each corresponding to an on-device inference engine. We then reorganize each model copy by implementing engine-specific operator optimizations. However, directly executing the extracted code snippets from *pattern mining* is infeasible, as the code typically depends on build configurations and environment-specific dependencies that are not included in the snippet. To tackle this issue, rather than simulating a complete execution environment, we prompt (P.3) the agent to rewrite the code snippet for operator optimization. Our prompt explicitly separates engine-specific details from optimization logic and enforces a graph-level abstraction so that the rewritten code can be consistently integrated into the model reorganization pipeline.

**P. 9: Code Rewriting Prompt (simplified)**

Given a code snippet that implements an operator-level optimization (fusion and removal) used by an inference engine, your task is to rewrite this snippet so that it becomes a standalone, portable function that can be directly applied to a computational graph of a model.

**Rules:** 1. Abstract away engine-specific dependencies

\* Remove references to build configs or environment-specific libraries.

\* Replace engine API calls with generic tensor/graph operations.

2. Preserve optimization semantics

\* Maintain the same operator transformation (*e.g.*, Conv+Add→Conv)

3. Code should be organized in a structured function

\* Input: a graph representation of the model.

\* Output: a modified graph with the optimization applied.

**Raw code segment:** {...}; **Optimizable pattern:** {...}.

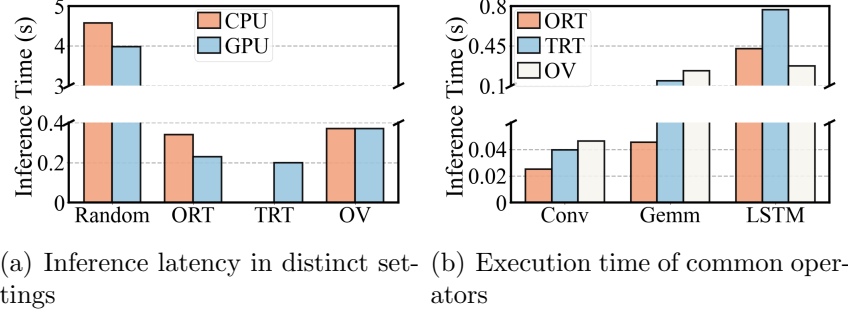


Figure 6.6: (a) Inference latency when randomly assigning compatible inference engines; (b) The three engines require distinct times to execute the three common operators (Gemm – general matrix multiplication).

Mathematically, the model reorganization process is expressed as:

$$\begin{aligned} \mathcal{M} &\xrightarrow{\text{reorganize}} \{m_1, m_2, \dots, m_k\}, \\ m_j &\sim \{\sigma_1^j, \dots, \{\sigma_r^j, \sigma_{r+1}^j, \dots\} \xrightarrow{\text{merge}} \sigma_r^j, \dots\} \end{aligned} \quad (6.9)$$

where  $\mathcal{M}$  is the user-specified pretrained model,  $k$  is the number of on-device inference engines and  $m_j$  is the  $j$ -th reorganized model containing a list of optimized operators.

### 6.4.3 Greedy Cross-Engine Inference Algorithm

Based on the obtained information of operator-engine compatibility and reorganized models, the final step is to apply model partitioning and perform cross-engine inference appropriately.

**Challenges.** A straightforward approach to cross-engine inference is to randomly select a compatible engine for each operator or pattern. Nevertheless, we find that it often results in increased inference time. Following the same experiment settings in § 6.4.1, we measure inference time in two scenarios: 1) randomly selecting a compatible engine for each operator and 2) using a single engine by manually registering all operators. As illustrated in Fig. 6.6(a), the former requires significantly longer

inference times than the others. A further breakdown uncovers two important factors:

- Even for the same operator, different engines require distinct computation times. Fig. 6.6(b) shows the execution time of the three SOTA engines across three common operators. We can see that OpenVINO executes **LSTM** more efficiently, while ONNX Runtime computes **Conv** faster. Thus, even for a single operator, engines exhibit significant performance variations, let alone when existing AI models contain hundreds of such basic operators.
- Random engine selection inevitably causes frequent context switching, *i.e.*, transferring intermediate outputs from one engine to the next each time a new engine is selected. The engine switching overhead further exacerbates inference time.

Consequently, randomly selecting a compatible inference engine for operator execution leads to suboptimal inference times.

**Solution & Rationale.** To address the above challenges, an intuitive approach would be to search all possible combinations of model partitions and inference engines. However, this method is prohibitively expensive and time-consuming. Instead, as demonstrated in Fig. 6.7, we propose a *greedy cross-engine inference algorithm*, with its pseudocode presented in Algorithm 1. The core idea is to sequentially assign each operator in the pretrained model to the engine that both supports it and maximizes optimization benefits while minimizing engine-switching overhead. Moreover, to prevent being trapped in local optimums, we introduce a *bounded backtracking* mechanism that reverts to the previous state if the current partition scheme results in a longer inference time. By jointly considering *operator-engine compatibility*, *optimizable operator patterns*, and *switching costs* at each step, we can incrementally make locally optimal decisions and progressively converge. This design substantially

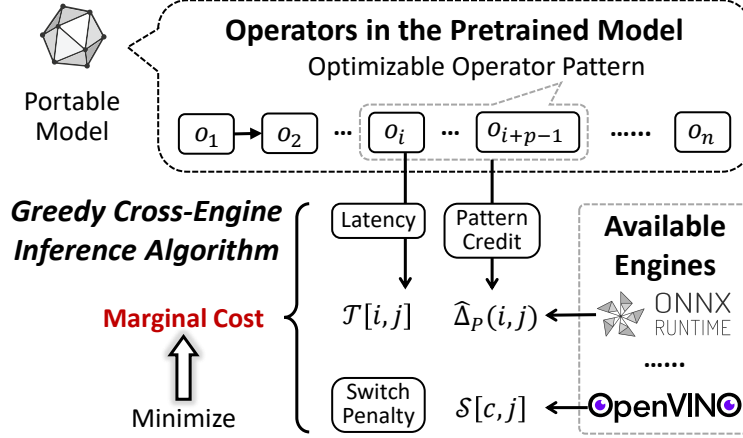


Figure 6.7: The *greedy cross-engine inference algorithm*.

reduces computational complexity, enabling faster convergence without the need to exhaustively explore the entire search space.

**Definitions & Initialization.** We begin by introducing relevant notations and their definitions. The pretrained model  $\mathcal{M}$  is composed of a list of operators  $\{o_1, o_2, \dots, o_n\}$ . The target embedded device has  $k$  on-device inference engines  $\{e_1, e_2, \dots, e_k\}$ . The operator-engine compatibility matrix is  $\mathcal{C}^{n \times k}$ , where  $\mathcal{C}[i, j] = 1$  means the operator  $o_i$  is supported by engine  $e_j$ . After model reorganization, we obtain  $k$  reorganized models  $\{m_1, m_2, \dots, m_k\}$  that contain  $k$  sets of engine-specific optimizable operator patterns. We leverage nn-Meter [227] to profile the execution time  $\mathcal{T}^{n \times k}$  of each operator in the pretrained model, as well as the engine switching time  $\mathcal{S}^{k \times k}$ .

**Pattern Saving Credit.** We first quantify the benefits of preserving each engine-specific optimizable operator pattern during cross-engine inference. Specifically, For an optimizable pattern  $\{o_i^j, o_{i+1}^j, \dots, o_{i+p-1}^j\}$  in  $m_j$  corresponding to engine  $e_j$ , we first define a small lookahead window  $w$  that, at operator index  $i$ , returns the set of patterns that complete within  $[i, i + w - 1]$  if the required operators are kept on  $e_j$ . The pattern saving credit  $\Delta(p, j)$  is then defined as the time saved by executing the

optimized operators (merged or removed) compared with executing these operators sequentially. Mathematically, this process can be expressed as:

$$\begin{aligned} \text{completions}_j(i, w) &\subseteq \{\sigma_i^j, \sigma_{i+1}^j, \dots, \sigma_{i+p-1}^j\} \\ \Delta(p, j) &= \sum_{r=i}^{i+p-1} \mathcal{T}_r^j - \mathcal{T}_{\hat{r}} \end{aligned} \quad (6.10)$$

where  $\mathcal{T}_{\hat{r}}$  is the execution time of the optimized operator. Furthermore, a discounted pattern credit is created to estimate the benefit unlocked by assigning  $o_i$  to  $e_j$ :

$$\begin{aligned} \hat{\Delta}_P(i, j) &= \sum_{p \in \text{completions}_j(i, w)} \gamma(p, i, j) \cdot \Delta(p, j) \\ \gamma(p, i, j) &= \begin{cases} \prod_{t=i}^{i+p-1} \exp(-\alpha \cdot \hat{X}(p, i, j)), & \forall t \Rightarrow \mathcal{C}[t, j] = 1 \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (6.11)$$

where  $\gamma(p, i, j)$  is the discount,  $\hat{X}(p, i, j)$  is a local estimate of extra switches that would be induced to keep the span on  $e_j$  (computed from recent assignments and  $\mathcal{S}$ ), and  $\alpha > 0$  is a discount factor.

**Greedy Scoring.** We then define a greedy scoring function that evaluates each operator-engine pair by balancing immediate execution latency, pattern credit, and engine-switching overhead to select the locally optimal inference engine at each step. For the first operator  $o_1$ , we assign it to the engine with the shortest execution time, denoted as  $e_c$  (*i.e.*, the current inference engine). Then, for each subsequent operator  $o_i$  in the pretrained model, we compute the **marginal cost** when assigning  $o_i$  to each compatible engine  $e_j$ , considering the three constraints:

$$\mathbf{MC}(i, j) = \underbrace{\mathcal{T}[i, j]}_{\text{latency}} - \underbrace{\lambda_P \cdot \hat{\Delta}_P(i, j)}_{\text{pattern credit}} + \underbrace{\lambda_S \cdot \mathbb{1}[e_j \neq e_c] \cdot \mathcal{S}[c, j]}_{\text{switch penalty}} \quad (6.12)$$

where  $\lambda_S$  and  $\lambda_P$  are two weights. We then pick the engine  $e_{j^*}$  that minimizes  $\mathbf{MC}(i, j)$ , tie-breaking in favor of the current engine.

**Bounded Backtracking.** Greedy decisions may become trapped in local optima [268], blocking the discovery of high-value optimizable patterns. Therefore, we de-

---

**Algorithm 1:** Greedy Cross-Engine Inference Algorithm
 

---

**Input :** Pretrained model  $\mathcal{M} = \{o_1, \dots, o_n\}$ ;  
 On-device inference engines  $\mathcal{E} = \{e_1, \dots, e_k\}$ ;  
 Compatibility matrix  $\mathcal{C}^{n \times k}$ ; Engine switching cost  $\mathcal{S}^{k \times k}$ ;  
 Operator latency matrix  $\mathcal{T}^{n \times k}$  (from nn-Meter);  
 Pattern libraries  $\{\mathcal{P}_1, \dots, \mathcal{P}_k\}$ ; Regret threshold  $\rho$ ;  
 Lookahead window  $w$ , backtrack depth  $B$ ;  
 Weights  $\lambda_S, \lambda_P$  and discount factor  $\alpha$ .

**Output:** Engine assignment list  $assign = [a_1, \dots, a_n]$ ;  
 Segment list  $segments$ ; total estimated cost  $total\_cost$ .

```

1 Initialize  $assign \leftarrow []$ ,  $segments \leftarrow []$ ,  $history \leftarrow []$ ;
2 Initialize  $total\_cost \leftarrow 0$ ,  $e_c \leftarrow \emptyset$ ;
3 for  $i = 1$  to  $n$  do
4    $candidates \leftarrow \{e_j \mid \mathcal{C}[i, j] = 1\}$ ;
5   foreach  $e_j \in candidates$  do
6      $MC \leftarrow \mathcal{T}[i, j]$ ;
7     if  $e_c \neq \emptyset$  and  $e_j \neq e_c$  then
8        $MC \leftarrow MC + \lambda_S \cdot \mathcal{S}[e_c, e_j]$ ;
9     end
10     $\hat{\Delta}_P \leftarrow 0$ ;
11    foreach  $p \in matchers[e_j].completions(i, w)$  do
12       $disc \leftarrow \gamma(p, i, j, \mathcal{C}, \mathcal{S}, \alpha)$ ;
13       $\hat{\Delta}_P \leftarrow \hat{\Delta}_P + disc \cdot \Delta(p, j)$ ;
14    end
15     $MC \leftarrow MC - \lambda_P \cdot \hat{\Delta}_P$ ;
16    Record  $(MC, e_j)$  in score list;
17  end
18  Select  $e^* = \arg \min_{e_j} MC(i, j)$  (tie-break in favor of  $e_c$ );
19   $assign[i] \leftarrow e^*$ ;
20  Update  $segments$  and  $total\_cost$ ;
21  Push  $(i, e_c, e^*, MC, \text{snapshot}(matchers))$  into  $history$ ;
22   $e_c \leftarrow e^*$ ;
23  if  $should\_backtrack(i, history, matchers, \rho, B)$  then
24     $i_{rev} \leftarrow \text{find\_revert\_point}(history, B)$ ;
25    if  $i_{rev} \neq \emptyset$  then
26       $\text{restore\_from}(history, i_{rev})$ ;
27       $\text{alternative\_at}(i_{rev}, assign, segments, \mathcal{C}, \mathcal{T}, \mathcal{S}, \mathcal{P})$ ;
28    end
29  end
30 end
31 return  $(assign, segments, total\_cost)$ ;

```

---

sign a *bounded backtracking* mechanism as a remedy that selectively reverts a limited number of recent engine assignments to recover potential optimizable patterns. Specifically, the algorithm records a short history of recent decisions, including operator indices, assigned engines, and marginal costs. After each assignment, the algorithm evaluates whether reverting a recent switch could unlock a high-value pattern in completions $_j(t, w)$  within a bounded window  $t \in [i - B + 1, \dots, i]$ . If the potential savings exceed a regret threshold  $\rho$ , *i.e.*, fraction of the recent window cost,

Table 6.2: Detailed information of the embedded devices.

| Type            | Device              | RAM   | VRAM | Storage |
|-----------------|---------------------|-------|------|---------|
| Embedded Device | Raspberry Pi 5      | 8GB   | -    | 32GB    |
|                 | Jetson Nano         | 4GB   | 4GB  | 64GB    |
|                 | IBOX-606 mini PC    | 32GB  | 12GB | 256GB   |
| Router          | Mi AX3000           | 256MB | -    | -       |
| Edge Server     | Supermicro X11SCA-F | 32GB  | 16GB | 1T      |

the algorithm rolls back to the corresponding decision, replaces the engine assignment with the optimal alternative, and replays greedily forward.

**Complexity Analysis.** In conventional brute-force search-based methods, each of the  $n$  operators in the pretrained model has  $k$  possible engine selections, resulting in  $O(k^n)$  distinct operator-engine combinations, which is highly inefficient. In contrast, our greedy traversal sequentially evaluates each operator once, considering at most  $W$  pattern completions within a fixed lookahead window  $w$ . Therefore, the per-operator computational cost is  $O(kW)$ , and the total complexity is  $O(nkW)$ . Since both  $W$  and the bounded backtracking depth  $B$  are small constants in practice, the algorithm achieves near-linear time complexity with respect to  $n$ . Detailed theoretical analysis could be found in § 9.1

## 6.5 Experiment Setup

### 6.5.1 Implementation

**Hardware.** We implement `AutoDeploy` on an edge server and deploy various models on three embedded devices: a Raspberry Pi, a Jetson Nano board, and an embedded mini PC. The edge server and the embedded devices are connected via a router within a local area network. We use CPU runtimes for models deployed on the Raspberry Pi and the mini PC, and GPU runtimes for those executed on the Jetson Nano board and the mini PC. Table 6.2 shows the hardware configurations of the devices.

**Software.** We implement `AutoDeploy` with 2,432 lines of Python code. `AutoDeploy` employs `LangChain` [91] as the agent development framework, `Tavily` [98] as the web search engine, and `Chroma` [239] as the vector database for storing the extracted operator knowledge. We further develop a communication interface for `AutoDeploy` to remotely access and control target embedded devices. `GPT-4o` [269] serves as the backend LLM of `AutoDeploy`, while `text-embedding-ada-002` [270] is used for text embedding generation. Note that we find that other LLMs, including open-sourced models, can also accurately perform compatibility checking and optimizable pattern detection.

## 6.5.2 Evaluation Protocols

We evaluate `AutoDeploy` using 150 pretrained models for a comprehensive assessment and three embedded AI applications for a case study. In addition, we conduct a user study to examine the *generalizability* of `AutoDeploy`, where we invite users to propose improvisational requirements for deploying diverse AI models.

**Pretrained Models.** As mentioned in § 6.2.2, we download 150 pretrained AI models from Hugging Face based on different model architectures, inference tasks, model sizes, and pretrained model formats. Detailed information on these models is shown in Table 6.2.

**Embedded AI Applications.** *Based on input data modality, model architecture, and device heterogeneity, we select three representative embedded AI applications: image recognition (visual), speech sentiment analysis (audio), and visual-language understanding (multimodal).* 1) **Image Recognition (IR)** on embedded devices enables real-time processing and analysis of *visual data* in areas such as autonomous driving [70, 271] and healthcare [69]. We select a pretrained *CNN*-based ResNet-152 [272] model, instruct `AutoDeploy` to deploy it on a *Raspberry Pi*, and evaluate it on a subset of the ImageNet dataset [273] that contains 100,000 RGB images in

1,000 classes. 2) **Speech Sentiment Analysis (SSA)** on *audio data* involves analyzing speech patterns to determine the emotional tone of spoken content, fostering smart voice assistants, customer service interactions, *etc.* We select the fine-tuned *CNN+Transformer*-based Whisper model [274], prompt **AutoDeploy** to deploy it on a *Jetson Nano* board, and assess it using the TESS [275] dataset that has 2,800 audio samples in seven emotion types. 3) **Vision-Language Understanding (VLU)** leverages *Vision-Language Models* (VLMs) to interpret and describe general images (*e.g.*, logical diagrams and formula recognition), enabling applications such as visually impaired assistant [276]. We select the pre-trained multimodal *Transformer*-based DeepSeek-VL-1.3b [277] as the VLM model, request **AutoDeploy** to deploy it on the *edge mini PC*, and evaluate it on the MMMU benchmark [241], comprising 11.5K meticulously collected image understanding questions covering six core disciplines, *i.e.*, art & design, business, science, medicine & healthcare, humanities, and technology & engineering. For all the models, we manually convert them into the four model types (PyTorch [256], ONNX [243], TensorFlow [47], and Keras [278]) for deployment, respectively. We also ensure that each model is affordable by the resource-constrained embedded devices. *Note that the fine-tuned Whisper model and the VLM use certain operators (e.g., LayerNormalization, FusedConv) that are not fully supported by any single inference engine.*

## 6.6 Evaluation

### 6.6.1 Metrics

To evaluate **AutoDeploy**, we adopt two metrics: 1) **Deployment success rate**. To investigate the efficacy of **AutoDeploy**, we record the *proportion* of models that can be successfully deployed and executed on the target embedded device. 2) **Wall-clock time**. To measure the efficiency of **AutoDeploy**, we record the *time* required from the

moment a user specifies a pretrained model to its successful deployment on the target embedded device. Furthermore, to assess the real-time performance of the deployed models, we adopt three types of metrics: 1) **Inference Latency**. We measure the *time* required to produce an inference result. 2) **Resource Consumptions**. We measure the maximum *CPU RAM usage* and *GPU VRAM usage* during model execution. 3) **Task Accuracy**. For IR and SSA, we record the *classification accuracy* to assess how effectively the on-device model recognizes objects from input images or identifies emotions from audio data. For VLU, we measure the *micro-averaged accuracy* as defined by the MMMU benchmark [241].

## 6.6.2 Baselines

To the best of our knowledge, no existing system in literature can automate model deployment for heterogeneous embedded devices. Thus, we consider three baseline types:

- **SOTA Inference Engines**: ONNX Runtime [255], TensorRT [45], OpenVINO [46], TVM [257], and MLIR [258]. Note that we manually register unsupported operators into these inference engines.
- **Human<sup>†</sup>** We invite 15 developers and evenly divide them into three groups based on their self-assessed development experience and expertise. **Human Group 1 (H1)** are experts with extensive experience in deploying AI models on various embedded devices. **Human Group 2 (H2)** are semi-experts who only have extensive experience in training AI models but with basic knowledge about embedded devices and model deployment. **Human Group 3 (H3)** are non-experts who have limited knowledge of AI models and embedded devices with little practical experience. *Note that the users are permitted to freely use*

---

<sup>†</sup>All human evaluations were approved by our institution’s ethics review board (IRB).

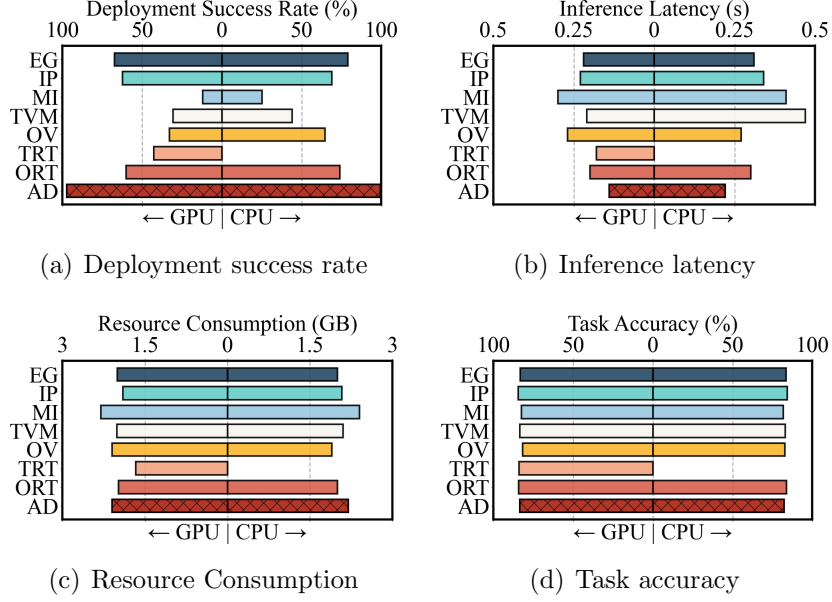


Figure 6.8: Evaluation results on the 150 open-source models (Left: GPU runtime; Right: CPU runtime; Labels from bottom to top: AD – AutoDeploy, ORT – ONNX Runtime, TRT – TensorRT, OV – OpenVINO, MI – MLIR, IP – IoT Pilot, EG – EmbedGenius).

*LLMs to assist their work.*

- **Agentic Coding Systems.** We select two SOTA agent-based coding systems that are tailored for embedded AI application development: IoT Pilot [6] and EmbedGenius [7]. We prompt each agent to synthesize an executable program for on-device model deployment and inference.

### 6.6.3 Comprehensive Model Evaluation

We first instruct AutoDeploy to deploy the 150 pretrained models across the three embedded devices, using both CPU and GPU runtimes, respectively. During model execution, we record the deployment success rate, the average inference latency, resource consumption, and task accuracy. As presented in Fig. 6.8, we find that:

- ❶ AutoDeploy achieves a near-100% deployment success rate, significantly

**outperforming the baselines.** This is primarily because `AutoDeploy` retrieves and analyzes operator-engine compatibility to ensure that every model operator is executable on a supported inference engine. The only failure case occurs with the “webpage\_element\_detection” model [279], as it uses the `Upsample` operator, which was deprecated by the selected inference engine (*i.e.*, ONNX Runtime) since operator version 10 in favor of `Resize` [280, 262]. In contrast, single-engine-based methods fail to execute at least 25.3% of the pretrained models on various embedded devices due to unsupported operators. Similarly, agent-based coding frameworks are unaware of the *model-engine mismatch* issue and tend to randomly select a single engine for on-device inference.

② **`AutoDeploy` also achieves lower inference latency.** The underlying reason is that `AutoDeploy` integrates multiple inference engines and organically harnesses their internal operator optimization methods for more efficient cross-engine inference. In addition, our greedy algorithm in `AutoDeploy` facilitates achieving an optimal balance among ensuring operator-engine compatibility, preserving optimizable operator patterns, and avoiding frequent context switching. On the contrary, agent-based coding systems randomly select a compatible engine for on-device inference, indicating that even with manually registered operators, adhering to a single engine may still lead to suboptimal inference latency.

③ **`AutoDeploy` maintains real-time performance comparable to the baselines without sacrificing task accuracy.** Although `AutoDeploy` simultaneously hosts multiple inference engines at runtime, its memory consumption is only slightly more than that of the baselines (just over 200 MB). This increase remains acceptable in light of the model size [281]. Moreover, `AutoDeploy` achieves an average task accuracy on par with that of the baselines.

The above results demonstrate that `AutoDeploy` can effectively address the *model-engine mismatch* issue by leveraging multiple inference engines to collectively support

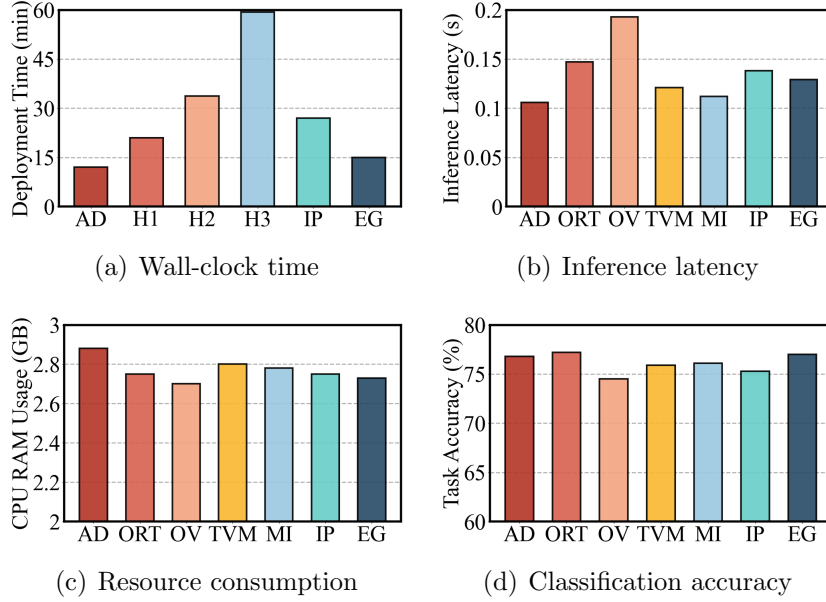


Figure 6.9: Evaluation results for image recognition (AD – **AutoDeploy**, H1 – human group 1, H2 – human group 2, H3 – human group 3, IP – IoTPIlot, EG – EmbedGenius, ORT – ONNX Runtime, OV – OpenVINO, MI – MILR).

a diverse set of model operators, meanwhile reducing overall inference latency and maintaining real-time performance comparable to the baselines.

#### 6.6.4 Case Study: Embedded AI Applications

**Image Recognition (IR).** We propose deploying the ResNet-152 model on a Raspberry Pi for image recognition and repeating the experiment 20 times to avoid randomness. From Fig. 6.9, we observe that **AutoDeploy** significantly reduces the time (only around 12 minutes) required for model deployment across different embedded devices. On the contrary, human baselines need substantially 79.3% more time to successfully deploy the IR model, even when they may resort to LLMs’ assistance. Additionally, agent-based assistants spend over 15 minutes iteratively refining the generated code to ensure that it is executable for model deployment. These results further demonstrate the effectiveness of **AutoDeploy** in translating user requirements to executable, on-device embedded models to enable efficient inference.

On top of reducing deployment time, **AutoDeploy** also achieves lower inference latency than the baselines, with an average reduction of approximately 21.9%. This performance gap widens (Fig. 6.10 and Fig. 6.11) as the model becomes more complex and includes a greater variety of unsupported or developer-customized operators. The underlying reason is that the *structure-aware pattern detector* identifies optimizable patterns that are exploited during cross-engine model inference, thereby reducing overall inference latency. Additionally, compared with the baselines, **AutoDeploy** achieves more stable real-time performance with lower standard deviations across all evaluation metrics. In particular, **AutoDeploy** achieves 1.3 minutes, 0.02 seconds, 0.05 GB, and 2.3% standard deviations in terms of deployment time, inference latency, CPU RAM usage, and task accuracy, respectively. Notably, this observation is consistent across all the 3 AIoT applications, and the benefit of **AutoDeploy** is amplified especially when the task becomes too complex.

Last but not least, **AutoDeploy** consumes around 220 MB more memory than the baselines while maintaining comparable classification accuracy.

**Speech Sentiment Analysis (SSA).** Fig. 6.10 shows the evaluation results of the tuned Whisper model after deployment. Since this model is more complex and contains more unsupported operators, human baselines require significantly longer deployment times, with non-experts even taking up to 70 minutes. In contrast, LLM-assisted systems, *i.e.*, **AutoDeploy**, IoTPilot, and EmbedGenius, require substantially less time for embedded Whisper deployment, benefiting from the automated workflows of AI agents. Notably, **AutoDeploy** consumes nearly half the deployment time of IoTPilot and EmbedGenius. This improvement arises because the model inference process is entirely performed via our efficient *greedy cross-engine inference algorithm*, whereas agents are used solely for operator analysis. This design reduces overall deployment time compared with the other two systems, which entirely rely on agents for model deployment code generation and iterative refinement.

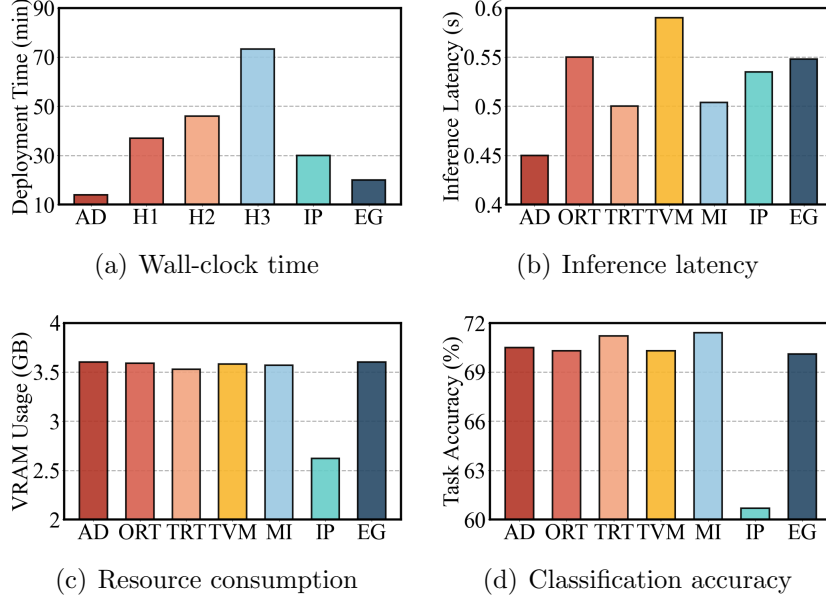


Figure 6.10: Evaluation results for speech sentiment analysis (SSA).

Interestingly, we found that the IoTPilot-deployed model consumes less memory but attains significantly lower accuracy than the others. Further investigation reveals that IoTPilot detects that some operators cannot be executed by any on-device inference engine and generates code to remove them. Consequently, the resulting model becomes smaller and consumes less memory but exhibits lower accuracy. Moreover, among the baselines, TensorRT achieves the lowest inference latency of approximately 0.5 ms. This is because TensorRT, developed by NVIDIA, is highly optimized for NVIDIA-designed hardware, *i.e.*, the Jetson Nano board, and its corresponding runtime environment.

**Vision Language Understanding (VLU).** Fig. 6.11 shows the evaluation results of the deployed VLU model. We observe that as the model size increases (the fine-tuned Whisper model contains 0.6B parameters, whereas the VLU model has 1.3B parameters), the efficiency of **AutoDeploy** in embedded model deployment becomes increasingly evident, with the wall-clock deployment time remaining around 15 minutes. Moreover, the time consumption of agent-based systems increases significantly

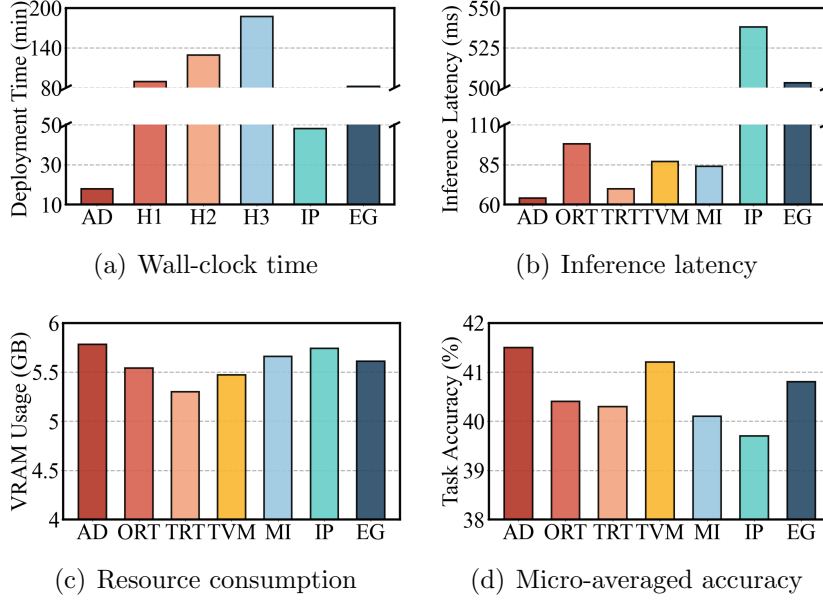


Figure 6.11: Evaluation results for vision-language understanding (VLU).

because they are unaware of the *model-engine mismatch* issue and repeatedly modify the model deployment code without clear guidance. As a result, they fall back to CPU runtimes for model inference, leading to substantially longer execution times (exceeding 500 ms). Similarly, **AutoDeploy** can also retain comparable real-time performance in terms of both resource consumption and task accuracy.

**Remarks.** Overall, the superior performance of the embedded models deployed by **AutoDeploy** stems from three key components: the *agent-based operator analyzer*, which automatically inspects operators with minimal user intervention and enhanced deployment efficiency; the *structure-aware pattern detector*, which preserves optimizable patterns to reduce inference latency; and the *greedy cross-engine inference algorithm*, which comprehensively accommodates multiple factors to optimize overall inference time.

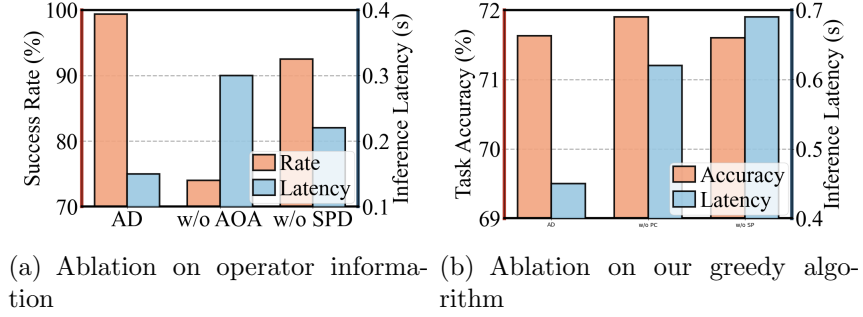


Figure 6.12: Ablation study on speech sentiment analysis (AOA – Agent-Based Operator Analyzer, SPD – Structure-Aware Pattern Detector, PC – pattern credit, SP – switching penalty).

### 6.6.5 Ablation Study

We conduct ablation studies to evaluate the impact of the technical modules in **AutoDeploy**. For demonstration purposes, we use speech sentiment analysis as an example, though similar phenomena are also observed in the other two embedded AI applications.

**Operator Information.** To evaluate the impact of the retrieved operator information, we conduct an ablation study with three configurations: 1) the full **AutoDeploy** system, 2) **AutoDeploy** without the *agent-based operator analyzer*, and 3) **AutoDeploy** without the *structure-aware pattern detector*. We report the average deployment success rate and inference latency in Fig. 6.12(a). We find that removing the operator analyzer results in a substantial drop in deployment success rate, as it provides critical compatibility information between each operator in the pretrained model and various on-device inference engines. Without this knowledge, **AutoDeploy** cannot assign each model partition to appropriate engines for effective on-device inference and often falls back to inefficient CPU runtimes. Moreover, disabling the pattern detector causes a dramatic increase in inference latency, as optimizable operator patterns are no longer preserved during partitioning.

**Greedy Algorithm.** To investigate the role of different factors in our greedy al-

Table 6.3: Participant information of the user study (N=20)

| Category         | Background Information                                          |
|------------------|-----------------------------------------------------------------|
| <b>Gender</b>    | Female (45%), Male (40%), Prefer not to say (15%)               |
| <b>Age</b>       | Under 18 (5%), 18-30 (75%), 30-39 (10%), 40 and older (10%)     |
| <b>Education</b> | Bachelor (5%), Master (20%), Doctoral (70%), Others (5%)        |
| <b>English</b>   | Beginner (5%), Intermediate (20%), Advanced (65%), Fluent (10%) |
| <b>Expertise</b> | Expert (50%), Semi-expert (50%)                                 |

gorithm when calculating the *marginal cost* in Eq. 6.12, we consider three ablation configurations: 1) the full **AutoDeploy** system, 2) **AutoDeploy** without the pattern credit term, and 3) **AutoDeploy** without the switch penalty term. We then present the inference latency and task accuracy of the deployed model in Fig. 6.12(b). It can be observed that model accuracy remains largely stable across all settings, as the parameters of the pretrained model are unchanged. Nonetheless, both factors significantly affect inference latency. In particular, the inference latency increases by 1.5x when optimizable operator patterns are not preserved and nearly doubles when engine-switching overhead is neglected.

### 6.6.6 User Study for Generalizability Evaluation

To assess the utility, overall user satisfaction, and generalizability of **AutoDeploy**, we conduct a user study involving 10 new expert users and 10 semi-expert users. *In order to test the generalizability of **AutoDeploy**, we did not limit the tasks. Instead, participants are explicitly asked to freely express their requirements for deploying various open-source models on embedded devices to evaluate system generalizability.* Expert users have rich experience in inference engines and have developed multiple embedded AI applications, while semi-expert users are proficient in model training but possess only basic knowledge of embedded devices and engines. Table 6.3 lists the detailed background information of the participants.

**Evaluation Metrics.** We ask users to rate **AutoDeploy** based on four metrics: 1)

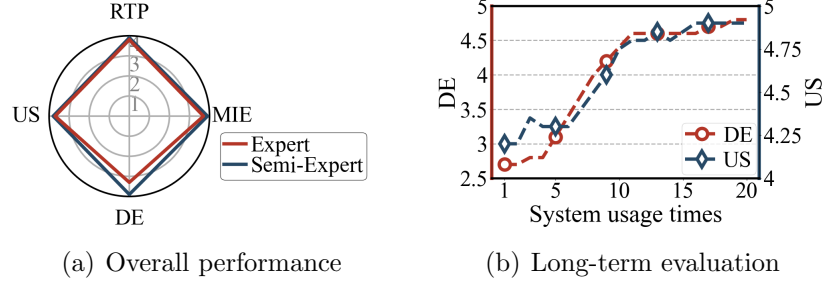


Figure 6.13: Evaluation results of the user study.

*Real-Time Performance (RTP)* evaluates the overall performance of the deployed model using private test data on the target device, including task accuracy and resource consumption; 2) *Model Inference Efficiency (MIE)* measures how quickly a model can produce an inference result given an input data sample; 3) *Deployment Efficiency (DE)* assesses how efficiently **AutoDeploy** deploys the optimized model on heterogeneous embedded devices for execution; 4) *User Satisfaction (US)* captures users' overall experience with **AutoDeploy** during usage. All metrics are rated on a scale from 1 (not at all) to 5 (completely).

**Overall Performance.** Fig. 6.13(a) illustrates the overall performance of the user study with averaged scores. Both experts and semi-experts provide high ratings (above 4.6) for **AutoDeploy** in terms of RTP, MIE, and US. However, semi-expert users tend to give higher scores than expert users for DE. This discrepancy stems from the semi-experts' limited familiarity with embedded model deployment, as well as their lack of domain knowledge regarding diverse inference engines and execution configurations tailored for heterogeneous embedded devices. As a result, they are more likely to rate higher scores for deployment efficiency since **AutoDeploy** provides an end-to-end solution from user-specified pretrained models to on-device efficient model inference across various embedded devices. *Such results exhibit **AutoDeploy**'s high generalizability in deploying various open-source models to heterogeneous embedded devices with enhanced user experience.*

**Long-term Evaluation.** We also invite expert users to participate in a long-term evaluation to investigate the impact on DE and US as system utilization increases. Fig. 6.13(b) shows that with increasing user engagement, both DE and US gradually increase, exhibiting a similar growth trend. As system usage time increases, *AutoDeploy* gradually retrieves and accumulates more comprehensive operator-engine compatibility information and optimizable operator patterns in the local database. Consequently, when new user requests arrive, our greedy algorithm can compute the marginal cost directly based on the database rather than repeatedly querying online sources. The local knowledge base is also continuously updated on a regular basis. As such, the deployment efficiency is enhanced as system utilization increases, and in turn, contributes to higher US scores. *These results demonstrate the outstanding usability and efficiency of **AutoDeploy** in long-term use.*

## 6.7 Discussion

**Agent-Assisted Operator Registration and Code Generation.** As AI agents rapidly progress, one might leverage them to automatically synthesize programs to register all unsupported model operators into inference engines. Though possible, our extensive experiments have demonstrated that sticking to a single inference engine still yields suboptimal inference latency (§ 6.6.3, § 6.6.4). Additionally, others may instruct AI agents (*e.g.*, EmbedGenius [7], MapCoder [1]) to generate code for cross-engine inference. However, after conducting 100 repeated experiments, we find that only 1% of the synthesized programs succeed, as these agents are designed for general-purpose programming and lack the domain knowledge required for embedded model deployment.

**Error Handling in Agentic Systems.** In agentic systems, a common challenge is the generation of incorrect intermediate results during multi-step reasoning. These

errors can propagate and lead to incorrect final outputs, significantly impacting agents’ reliability. To address this, **AutoDeploy** incorporates Reflexion [245] that enables agents to reflect on failed attempts and iteratively improve. Moreover, to further enhance the robustness of **AutoDeploy** in helping agents detect and recover from mistakes during multi-step reasoning, we plan to orthogonally integrate more advanced techniques (*e.g.*, ReAct [282]) as part of our future work.

## 6.8 Related Work

**Model Deployment on Embedded Devices.** Considering that embedded devices have constrained resources and limited computational capabilities, developers typically select an appropriate inference engine to deploy AI models based on specific hardware configurations and runtime environments of the device. Inference engines, effectively model accelerators, incorporate various internal optimization methods to enhance inference efficiency. While existing works [283, 284, 285] focus on enhancing model performance (*e.g.*, accuracy, latency) across heterogeneous execution environments [286, 287, 288], most of them ignore the *model-engine mismatch* issue. To address this problem, instead of relying on a single inference engine, **AutoDeploy** seeks to strategically partition models into several segments and leverage multiple inference engines to execute them alternately, meanwhile reducing overall inference latency.

**AI agents.** AI agents [56, 289, 290, 291, 292, 264] automate complex tasks by integrating LLMs [293, 294, 295, 296, 297] with various third-party tools (*e.g.*, web search engines). These agents can be applied to various applications, such as program synthesis [5, 298] and smart assistants [299]. Furthermore, some studies [2, 300] construct multiple agents to collaboratively accomplish complex tasks, allowing specialized agents to work together and adapt to dynamic environments efficiently. For

example, MapCoder [1] solves competitive programming tasks via four agents that handle different development stages. However, as discussed in § 6.7, these general coding agents are unaware of the *model-engine mismatch issue* and thus struggle to deploy diverse models on heterogeneous embedded devices (only 1% of success rate). In contrast, **AutoDeploy** features specialized design in operator analysis and operator-engine compatibility check, addressing unique technical challenges (*e.g.*, distinct operator names, preserving optimizable patterns) and significantly reducing human effort with enhanced deployment efficiency.

## 6.9 Chapter Summary

This paper presents **AutoDeploy**, an end-to-end system for automated AI model deployment on heterogeneous embedded devices. Targeting the *model-engine mismatch* issue, **AutoDeploy** addresses three key challenges, substantially improving the efficiency of embedded model deployment and reducing inference latency by strategically coordinating and scheduling multiple inference engines. In addition, user studies demonstrate **AutoDeploy**'s high generalizability in deploying various open-source models to heterogeneous embedded devices. We believe **AutoDeploy** can unlock the potential to drive the practical deployment of AI models across diverse and distributed embedded devices in real-world everyday applications.

# Chapter 7

## Discussions

To properly contextualize the contributions of this framework, it is necessary to examine the complete end-to-end AIoT application lifecycle, which fundamentally consists of four distinct stages: data collection via physical sensors, data transmission across networks, data processing using advanced algorithms and AI models, and decision-making or physical actuation. The empirical evaluations and system designs presented in this thesis, spanning `AutoIOT`, `GPIOt`, `SE2`, and `AutoDeploy`, predominantly emphasize the data processing and model deployment phases. We intentionally prioritized these specific stages because they represent the most severe technical bottlenecks for developers; they require complex, adaptive algorithms to handle fluctuating sensor noise, rigid memory constraints, and highly heterogeneous edge hardware runtimes. In contrast, the data collection and transmission stages are frequently managed by standardized network protocols and manufacturer-provided commercial off-the-shelf sample code. Nevertheless, the core agentic architecture we developed is fundamentally generalizable to the broader lifecycle. Because the proposed systems rely on natural language prompts, open-ended reasoning, and adaptable workflows, users can readily instruct these agents to synthesize scripts for initial raw data collection or cross-network data transmission. Moving forward, expanding this agentic paradigm to comprehensively automate and organically integrate

all four stages, such as deploying multiple collaborating agents to seamlessly handle cross-device synchronization alongside intelligent data processing, represents a highly promising direction for future autonomous AIoT research.

To properly contextualize the experimental evaluations presented in this thesis, it is important to address the scope of the selected AIoT application scenarios. Initially, evaluating automated code-generation and agentic systems in the AIoT domain presented a unique challenge: the research community inherently lacked comprehensive, high-fidelity benchmarks specifically designed to assess programming capabilities for sensor-based and embedded systems. Consequently, my early evaluations, such as those conducted for **AutoIoT**, were necessarily constrained to a curated set of representative case studies, including heartbeat detection and human activity recognition. To systematically overcome this community-wide limitation, we explicitly constructed **IoT Bench** during the development of **GPIoT**, introducing a dedicated benchmark tailored for evaluating AIoT program synthesis. In my subsequent research, including **SE<sup>2</sup>**, we transitioned to a more robust, two-tiered evaluation methodology: first leveraging **IoT Bench** to rigorously evaluate the code-generation capabilities of my proposed systems against state-of-the-art baselines across a broad spectrum of standardized tasks, and subsequently complementing these benchmark results with deep-dive analyses on three to four highly complex, real-world multimodal applications (such as Vision-Language Understanding and Object Detection in Autonomous Driving). This progression not only validates the adaptability of the frameworks proposed in this thesis but also establishes a foundational benchmarking standard for future AIoT software engineering research.

While this dissertation fundamentally adopts an empirical, systems-building methodology to solve practical AIoT engineering challenges, the proposed frameworks are strictly grounded in formal computational theory and mathematical optimization. For the physical deployment phase, the automated model partitioning

problem is formalized as a constrained graph optimization task, where the computation graph  $G = (V, E)$  must be mapped across heterogeneous engines. To solve this, the greedy cross-engine inference algorithm in **AutoDeploy** is designed with rigorous theoretical bounds; it achieves a near-linear time complexity of  $O(nkW)$  relative to the number of operators  $n$ , inference engines  $k$ , and lookahead window  $W$ , mathematically avoiding the intractable  $O(k^n)$  exponential complexity inherent to brute-force combinations. Furthermore, in the context of agentic programming, the dynamic workflow construction in **SE<sup>2</sup>** theoretically mirrors a Markov Decision Process (MDP), where the coordinator agent navigates a discrete state-action space (retrieve, reason, code, test) driven by intermediate semantic rewards derived from runtime feedback. Finally, the localized intelligence paradigms, specifically Parameter-Efficient Co-Tuning (PECT) in **GPIoT** and Reinforcement Fine-Tuning (RFT) in **SE<sup>2</sup>**, rely on rigorous policy-gradient mathematical updates to optimize agentic reasoning and minimize cross-domain misalignment. By formally bounding computational complexities and mathematically defining the state-action spaces of these multi-agent systems, this framework establishes a robust theoretical foundation that strongly complements its extensive empirical validation.



# Chapter 8

## Conclusions

The convergence of AIoT represents a transformative leap toward a world where physical devices possess the intelligence to perceive, reason, and act. However, the realization of this vision has long been hindered by a high technical barrier to entry. Developing robust AIoT applications traditionally demands “full-stack” expertise, spanning signal processing, deep learning architecture, and embedded systems optimization, creating a chasm between domain experts and deployable solutions. This thesis has presented a comprehensive framework for **Agentic Automated AIoT Application Development**, leveraging the emergent reasoning capabilities of LLMs to bridge this gap. By moving beyond simple code generation to autonomous, feedback-driven agentic workflows, we have demonstrated that it is possible to automate the entire lifecycle of an AIoT application, from natural language programming to optimized deployment on heterogeneous edge hardware.

To address the fundamental challenge of translating high-level human intent into executable embedded code, we introduced **AutoIOT**. This system established the first end-to-end framework capable of coupling LLMs with structure-aware knowledge retrieval and a novel compiler-feedback loop. By empowering non-experts to develop complex sensing applications using only natural language, **AutoIOT** effectively treats natural language as a new programming interface for the IoT. Recognizing that re-

liance on cloud-based automation introduces critical privacy risks and latency issues, we subsequently proposed **GPIO**T, a paradigm shift toward local intelligence. We demonstrated that Small Language Models (SLMs) can rival massive cloud models when tailored specifically for the IoT domain through our *Parameter-Efficient Co-Tuning (PECT)* strategy. This approach resolves the “domain misalignment” between task planning and coding, enabling collaborative SLMs to synthesize privacy-preserving code directly on the edge.

Furthering **AutoIoT** and **GPIO**T, to address the rigidity of traditional agentic automation pipelines, we introduced **SE**<sup>2</sup> to enable adaptive evolution. By moving beyond static workflows to a dynamic *Self-Exploration and Self-Evolution* paradigm, we created agents capable of constructing their own workflows on the fly. These agents leverage reinforcement learning and mobile runtime feedback to adapt to unseen sensors and changing environmental conditions, ensuring that automation remains robust in the chaotic physical world.

Last but not least, beyond code generation, this thesis tackled the persistent “model-engine mismatch” that bottlenecks the deployment of trained models to diverse edge devices. Through **AutoDeploy**, we replaced labor-intensive manual operator patching with an intelligent agentic workflow. By employing agents to analyze operator source code and utilizing a greedy cross-engine inference algorithm, the system automatically partitions models across available inference engines, significantly reducing inference latency and liberating developers from the burden of hardware-specific optimization.

Looking forward, the research presented in this thesis opens several promising avenues for future exploration. While our current frameworks focus on single-device applications, the next frontier involves multi-agent collaboration, where distinct agents representing different devices coordinate to build large-scale distributed systems, such as smart city infrastructures or swarm robotics [301]. Moreover, we expect an “Agent-

Everywhere” paradigm for end-to-end AIoT application development (from hardware setup to final deployment and maintenance). Agents can reside not only on our PC for vibe coding, but also even on heterogeneous mobile devices for grounded, runtime monitoring and deployment. Furthermore, extending our adaptive agents to support *lifetime learning* would allow deployed systems to continuously optimize themselves as sensors degrade or user behaviors shift, without requiring human intervention. Finally, as we grant agents greater autonomy to control physical actuators, developing rigorous verification frameworks to guarantee the safety of agent-generated code becomes paramount. Collectively, the contributions of this thesis establish a solid foundation for the next generation of AIoT developmentone that is automated, accessible, private, and continuously adaptive.



# Chapter 9

## Appendix

### 9.1 Complexity Analysis

This section provides a theoretical analysis of the proposed greedy cross-engine inference algorithm in Algorithm 1.

**Theorem 1** (Conditional Near-Optimality). *Let  $A_G = (a_1, \dots, a_n)$  denote the engine assignment produced by the proposed greedy cross-engine inference algorithm, and let  $A^* = (a_1^*, \dots, a_n^*)$  denote an optimal feasible assignment. The actual inference latency of an assignment  $A$  is defined as*

$$\mathcal{L}(A) = \sum_{i=1}^n \mathcal{T}[i, a_i] + \sum_{i=2}^n \mathbb{I}[a_{i-1} \neq a_i] \mathcal{S}[a_{i-1}, a_i] - \mathcal{G}(A), \quad (9.1)$$

where  $\mathcal{G}(A)$  denotes the total latency reduction obtained from the engine-specific operator optimization patterns preserved by  $A$ .

Assume that:

1. Every optimizable operator pattern  $p \in \mathcal{P}_j$  has length at most  $L$ :

$$|p| \leq L. \quad (9.2)$$

2. The lookahead window and backtracking depth satisfy

$$w \geq L, \quad B \geq L. \quad (9.3)$$

3. The estimated pattern benefit is conservative, i.e.,

$$0 \leq \widehat{\Delta}_P(i, j) \leq \Delta_P(i, j), \quad (9.4)$$

where  $\Delta_P(i, j)$  denotes the realizable optimization benefit associated with assigning  $o_i$  to  $e_j$ . The relative estimation error is bounded by  $\epsilon \in [0, 1)$ .

4. For every local window  $W$  containing at most  $B$  consecutive operators, the residual optimization benefit obtained by the optimal assignment but not preserved by the greedy assignment after the backtracking procedure terminates is bounded by

$$R(W) \leq (\rho + \epsilon) \mathcal{L}_G(W), \quad (9.5)$$

where  $\mathcal{L}_G(W)$  denotes the actual inference latency of the greedy assignment within  $W$  and  $\rho + \epsilon < 1$ .

Then the proposed greedy algorithm satisfies

$$\boxed{\mathcal{L}(A_G) \leq \frac{1}{1 - \rho - \epsilon} \mathcal{L}(A^*)}. \quad (9.6)$$

*Proof.* We first establish that the assignment produced by the algorithm is feasible.

At iteration  $i$ , the algorithm constructs the candidate engine set

$$\mathcal{E}_i = \{e_j \in \mathcal{E} \mid \mathcal{C}[i, j] = 1\}. \quad (9.7)$$

The selected engine  $a_i$  is always chosen from  $\mathcal{E}_i$ . Consequently,

$$\mathcal{C}[i, a_i] = 1, \quad \forall i \in \{1, \dots, n\}, \quad (9.8)$$

and therefore  $A_G$  is a feasible engine assignment.

For each candidate engine  $e_j \in \mathcal{E}_i$ , the algorithm evaluates the marginal cost

$$\text{MC}(i, j) = \mathcal{T}[i, j] + \lambda_S \mathbb{I}[e_j \neq e_c] \mathcal{S}[e_c, j] - \lambda_P \widehat{\Delta}_P(i, j), \quad (9.9)$$

where  $e_c$  is the engine assigned to the preceding operator.

The first term represents the execution latency of the current operator. The second term explicitly accounts for the context-switching overhead incurred when the current operator is assigned to an engine different from  $e_c$ . The third term rewards an engine assignment that preserves engine-specific optimization opportunities.

Consider an optimizable pattern

$$p = (o_s, o_{s+1}, \dots, o_t) \quad (9.10)$$

that can be optimized by engine  $e_j$ . By the bounded-pattern-length assumption,

$$t - s + 1 \leq L. \quad (9.11)$$

Since  $w \geq L$ , the complete pattern is contained in the lookahead window when the algorithm makes the decision for its first operator. Therefore, the pattern can be identified by

$$\text{matchers}[e_j].\text{completions}(i, w).$$

The corresponding estimated optimization benefit is

$$\hat{\Delta}_P(i, j) = \sum_{p \in \text{completions}(i, w)} \gamma(p, i, j, \mathcal{C}, \mathcal{S}, \alpha) \Delta(p, j). \quad (9.12)$$

By the conservative estimation assumption,

$$0 \leq \hat{\Delta}_P(i, j) \leq \Delta_P(i, j). \quad (9.13)$$

Hence, the greedy algorithm does not assign an optimization benefit larger than the benefit that can actually be realized.

The switching overhead is explicitly included in the marginal cost. For  $e_j \neq e_c$ , the additional cost is

$$\lambda_S \mathcal{S}[e_c, j], \quad (9.14)$$

whereas no switching cost is incurred when  $e_j = e_c$ . Thus, the greedy selection jointly considers the immediate execution latency, the optimization opportunities exposed by the current assignment, and the cost of switching from the current inference engine.

We next consider the effect of greedy decisions on optimizable patterns. Let

$$p = (o_s, \dots, o_t) \tag{9.15}$$

be a pattern that is preserved by  $A^*$  but is not preserved by  $A_G$ . Because

$$|p| \leq L \leq B, \tag{9.16}$$

the entire pattern lies within the backtracking horizon whenever a recent greedy decision invalidates it.

Suppose that the greedy assignment breaks the pattern at some operator  $o_r$ , where  $s \leq r \leq t$ . The backtracking procedure examines recent decisions through

$$\text{find\_revert\_point}(\text{history}, B)$$

and restores the state to a previous decision whenever

$$\text{should\_backtrack}$$

detects sufficient local regret. The subsequent

$$\text{alternative\_at}$$

operation evaluates an alternative feasible engine assignment and continues the greedy construction.

By the bounded-regret assumption, after this backtracking procedure terminates, the optimization benefit of patterns preserved by  $A^*$  but not preserved by  $A_G$  within any local window  $W$  satisfies

$$R(W) \leq (\rho + \epsilon)\mathcal{L}_G(W). \tag{9.17}$$

Partition the operator sequence into disjoint windows

$$W_1, W_2, \dots, W_q,$$

where every window contains at most  $B$  consecutive operators. Let  $R$  denote the total optimization benefit obtained by the optimal assignment but missed by the greedy assignment. Then

$$R = \sum_{r=1}^q R(W_r) \tag{9.18}$$

$$\leq (\rho + \epsilon) \sum_{r=1}^q \mathcal{L}_G(W_r) \tag{9.19}$$

$$= (\rho + \epsilon) \mathcal{L}(A_G). \tag{9.20}$$

Therefore,

$$\boxed{R \leq (\rho + \epsilon) \mathcal{L}(A_G)}. \tag{9.21}$$

We now compare the greedy assignment with the optimal assignment. Since the greedy objective explicitly accounts for the execution latency and switching overhead, the remaining difference between the two assignments is bounded by the optimization benefit that the optimal assignment obtains but the greedy assignment fails to preserve. Consequently,

$$\mathcal{L}(A_G) \leq \mathcal{L}(A^*) + R. \tag{9.22}$$

Substituting the bound on  $R$  gives

$$\mathcal{L}(A_G) \leq \mathcal{L}(A^*) + (\rho + \epsilon) \mathcal{L}(A_G). \tag{9.23}$$

Rearranging,

$$\mathcal{L}(A_G) - (\rho + \epsilon) \mathcal{L}(A_G) \leq \mathcal{L}(A^*), \tag{9.24}$$

$$(1 - \rho - \epsilon) \mathcal{L}(A_G) \leq \mathcal{L}(A^*). \tag{9.25}$$

Since  $\rho + \epsilon < 1$ , division by  $1 - \rho - \epsilon$  yields

$$\mathcal{L}(A_G) \leq \frac{1}{1 - \rho - \epsilon} \mathcal{L}(A^*). \quad (9.26)$$

Therefore, the proposed greedy cross-engine inference algorithm achieves an approximation ratio of

$$\boxed{\frac{1}{1 - \rho - \epsilon}}. \quad (9.27)$$

□

**Corollary 1** (Exact Pattern Estimation). *If the pattern optimization benefits can be estimated exactly, then  $\epsilon = 0$ . The approximation guarantee becomes*

$$\boxed{\mathcal{L}(A_G) \leq \frac{1}{1 - \rho} \mathcal{L}(A^*)}. \quad (9.28)$$

Moreover,

$$\lim_{\rho \rightarrow 0} \frac{1}{1 - \rho} = 1. \quad (9.29)$$

*Thus, when the pattern benefits are accurately estimated and the residual local regret approaches zero, the performance of the greedy algorithm approaches the global optimum.*

**Corollary 2** (Pattern Length and Backtracking Depth). *Suppose every optimizable pattern satisfies*

$$|p| \leq L. \quad (9.30)$$

*If*

$$w \geq L \quad \text{and} \quad B \geq L, \quad (9.31)$$

*then every optimization pattern affected by a recent engine assignment is observable within the lookahead window and remains within the backtracking horizon. Therefore,*

*the algorithm can reconsider any recent assignment that prevents the realization of an individual optimizable pattern.*

**Remark 1** (Nature of the Guarantee). *The above result is a conditional near-optimality guarantee rather than an unconditional approximation guarantee for arbitrary pattern libraries. A general greedy algorithm with finite lookahead and bounded backtracking cannot guarantee a constant approximation ratio for arbitrary combinations of operator compatibility, pattern interactions, and switching costs.*

*In particular, without the bounded-regret assumption, it is possible to construct an instance in which a locally optimal engine assignment prevents a high-value optimization pattern appearing farther in the model, while the pattern is outside the effective backtracking horizon. In such an instance, the performance ratio between the greedy solution and the global optimum can become arbitrarily large.*

*The parameter  $\rho$  therefore characterizes the residual local regret of the proposed algorithm, while  $\epsilon$  characterizes the error in estimating the optimization benefits of engine-specific patterns. The result establishes that, provided these two quantities remain bounded, the performance degradation relative to the global optimum is also bounded.*

## 9.2 Edge AI User Requirements

### 9.2.1 Multimodal HAR User Requirement

#### U. 1: Multimodal HAR Development

Implement a multimodal human activity recognition system using the XRF55 dataset (WIFI, RFID, and mmWave radar). The system should be deployed on a Raspberry Pi for on-device inference.

---

##### ## Dataset Information

The dataset folder is './Dataset/HAR/', containing: 'WIFI\_data.npy', 'RFID\_data.npy', 'mmWave\_data.npy', and 'label.npy'.

---

##### ## Functional Requirements

1. The system should use 'PyTorch' for model training.
2. The system should be optimized to run efficiently on a Raspberry Pi, taking advantage of its GPU for accelerated processing.

---

##### ## Performance Requirements after Deployment

1. Minimum accuracy on the testing dataset: 60%.
2. Maximum RAM usage during inference: 4GB.
3. Maximum inference latency per sample: 100ms.

---

##### ## Access the Raspberry Pi

1. The IP address of the Raspberry Pi: 192.168.31.100
2. Make a POST request in file mode to transmit scripts to the device.
  - URL: 'http://192.168.31.100:8080/script\_transmission'
  - Data: "script\_name": "script.py"
3. Make a POST request to execute the script and obtain the result.
  - URL: 'http://192.168.31.100:8080/model\_execution'
  - Data: "command": "python3 script.py -dataset\_path ; log.txt"
4. The testing dataset folder is '~/.Dataset/HAR/testing/', containing: 'WIFI\_data.npy', 'RFID\_data.npy', 'mmWave\_data.npy', 'label.npy'.

## 9.2.2 Multimodal VLU User Requirement

### U. 2: Multimodal VLU Development

Implement a Vision-Language Understanding (VLU) system that integrates images and text for image Q&A tasks. The system should be deployed on Jetson Nano for on-device inference.

#### ## Dataset Information

The dataset folder is './Dataset/VLU/', containing: 'image\_data.npy', 'text\_data.txt', 'answer.txt'

#### ## Functional Requirements

1. The system should use 'Pytorch' for model training.
2. The system should be optimized to run efficiently on a Jetson Nano, taking advantage of its GPU for accelerated processing.

#### ## Performance Requirements after Deployment

1. Minimum accuracy on the testing dataset: 35%.
2. Maximum RAM usage during inference: 4GB.
3. Maximum inference latency per sample: 100ms.

#### ## Access the Jetson Nano

1. The IP address of the Jetson Nano board: 192.168.31.100
2. Make a POST request in file mode to transmit scripts to the device.
  - URL: 'http://192.168.31.100:8080/script\_transmission'
  - Data: "script\_name": "script.py"
3. Make a POST request to execute the script and obtain the result.
  - URL: 'http://192.168.31.100:8080/model\_execution'
  - Data: "command": "python3 script.py -dataset\_path ./ log.txt"
4. The testing dataset folder is: './Dataset/VLU/testing/', containing: 'image\_data.npy', 'text\_data.txt', 'answer.txt'.

## 9.2.3 Multimodal ODAD User Requirement

### U. 3: Multimodal ODAD Development

Implement a multimodal object detection system for autonomous driving (cameras, radar, LiDAR, and GPS sensors). The system will be deployed on a laptop with a 16 GB GPU for on-device inference.

---

#### ## Dataset Information

The dataset folder is './Dataset/ODAD/', containing: 'camera\_data.npy', 'radar\_data.npy', 'lidar\_data.npy', 'gps\_data.npy', 'label.npy'.

---

#### ## Functional Requirements

1. The model should be implemented using PyTorch.
2. The system should be optimized to efficiently utilize the GPU, considering the available 16 GB memory.

---

#### ## Performance Requirements after Deployment

1. Minimum Average Precision: 50%
2. Maximum GPU memory usage: 14 GB
3. Maximum inference latency per sample: 1000ms.

---

#### ## Access the Laptop

1. The IP address of the laptop: 192.168.31.100
2. Make a POST request in file mode to transmit scripts to the device.
  - URL: 'http://192.168.31.100:8080/script\_transmission'
  - Data: "script\_name": "script.py"
3. Make a POST request to execute the script and obtain the result.
  - URL: 'http://192.168.31.100:8080/model\_execution'
  - Data: "command": "python3 script.py -dataset\_path ; log.txt"
4. The testing dataset folder is '~/.Dataset/ODAD/testing/', containing: 'camera\_data.npy', 'mmWave\_data.npy', 'audio\_data.npy', 'camera\_label.npy', 'mmWave\_label.npy', 'audio\_label.npy'.

## 9.3 Agentic Workflow Comparison

### 9.3.1 Baseline Workflows

The baselines attempt to design fixed, one-size-fits-all agentic workflows to accommodate diverse program synthesis tasks. However, these rigid workflows often suffer from inflexibility, limited adaptability, and high maintenance costs.

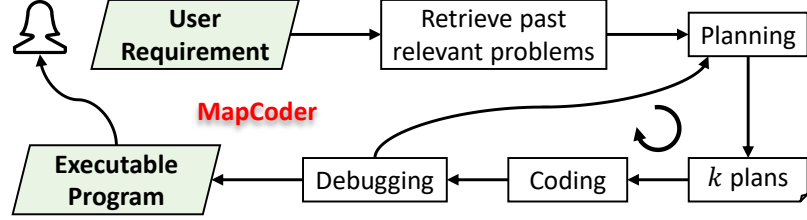


Figure 9.1: The agentic workflow of MapCoder [1].

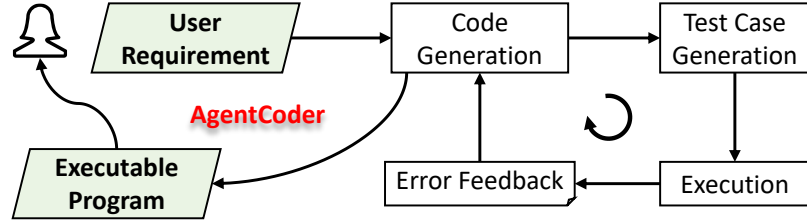


Figure 9.2: The agentic workflow of AgentCoder [2].

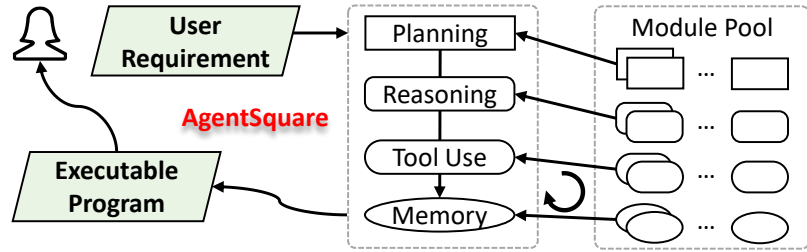


Figure 9.3: The agentic workflow of AgentSquare [3].

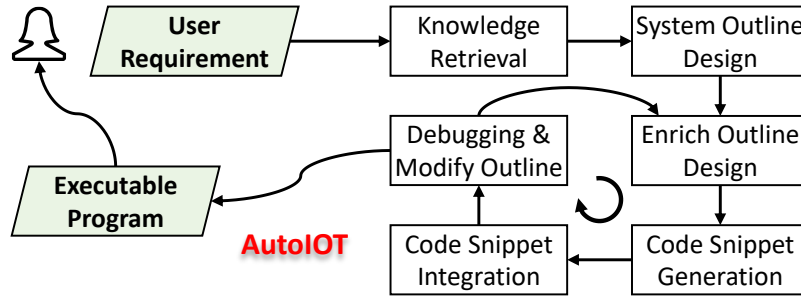


Figure 9.4: The agentic workflow of AutoIOT [4].

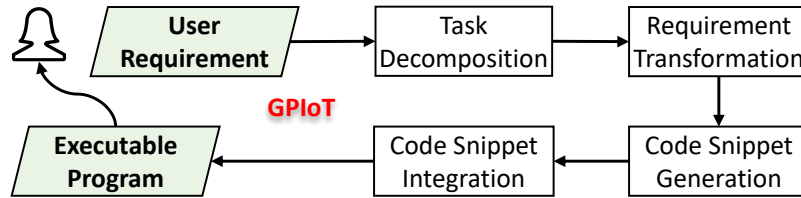


Figure 9.5: The agentic workflow of GPIoT [5].

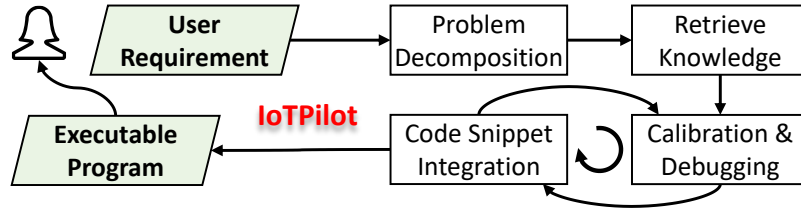


Figure 9.6: The agentic workflow of IoTPIlot [6].

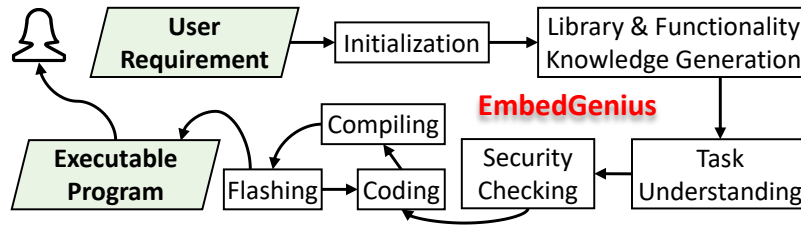


Figure 9.7: The agentic workflow of EmbedGenius [7].

### 9.3.2 SE<sup>2</sup> Workflow for HAR

The right table shows the autonomously constructed agentic workflow generated by SE<sup>2</sup> for multimodal HAR. Specifically, SE<sup>2</sup> first retrieves relevant domain knowledge and then generates distinct scripts to obtain the data dimensionality of each modality. After that, SE<sup>2</sup> creates tailored encoders for each modality and performs iterative debugging to ensure that each encoder can successfully process the input data. Then, SE<sup>2</sup> performs multimodal fusion by concatenating the encoder-extracted features and feeds the fused feature into a classifier model. Further, SE<sup>2</sup> collaboratively trains the encoders and the classifier with iterative debugging and code refinement. Finally, SE<sup>2</sup> deploys the trained model on the target device for inference.

| No.                                                         | Operation       | Instruction                                                                                                                                                                                |
|-------------------------------------------------------------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                                                           | <b>retrieve</b> | Search for human activity recognition.                                                                                                                                                     |
| 2                                                           | <b>retrieve</b> | Search for Wi-Fi CSI, mmWave, RFID.                                                                                                                                                        |
| 3                                                           | <b>reason</b>   | I need to first understand the data dimension.                                                                                                                                             |
| 4                                                           | <b>retrieve</b> | Search how to load '.np' file in Python.                                                                                                                                                   |
| 5                                                           | <b>code</b>     | Write a script to load the dataset and print out the data dimensionality.                                                                                                                  |
| 6                                                           | <b>test</b>     | Execute the code and analyze the data dimension.                                                                                                                                           |
| 7                                                           | <b>reason</b>   | Then, I need to design three encoders for each modality to extract effective features.                                                                                                     |
| 8                                                           | <b>retrieve</b> | Search some SOTA encoders for Wi-Fi CSI data.                                                                                                                                              |
| 9                                                           | <b>code</b>     | Based on the retrieved knowledge, write a script to build an encoder for the Wi-Fi data. Each data sample has a shape of (9, 30, 1000).                                                    |
| 10                                                          | <b>reason</b>   | Now I need to test whether the script can successfully process the input Wi-Fi CSI data.                                                                                                   |
| 11                                                          | <b>test</b>     | Generate some test cases that input a small batch of the Wi-Fi dataset into the encoder to test whether the script can be successfully executed.                                           |
| 12                                                          | <b>reason</b>   | The script failed because the encoder uses <code>Conv3d</code> to process the input data. But actually it should be <code>Conv2d</code> , where the input channel number is 9.             |
| 13                                                          | <b>code</b>     | Modify the previous script by changing the <code>Conv3d</code> to <code>Conv2d</code> where the input channel number is 9.                                                                 |
| 14                                                          | <b>test</b>     | Generate some test cases that input a small batch of the Wi-Fi dataset into the encoder to test whether the script can be successfully executed.                                           |
| 15                                                          | <b>reason</b>   | Now I need to create an encoder to extract features from the mmWave data.                                                                                                                  |
| Similar process of encoder design for mmWave and RFID data. |                 |                                                                                                                                                                                            |
| 27                                                          | <b>reason</b>   | Based on the three encoders, I need to perform feature fusion by concatenating all extracted features.                                                                                     |
| 28                                                          | <b>retrieve</b> | Search some SOTA multimodal fusion methods.                                                                                                                                                |
| 29                                                          | <b>code</b>     | Based on the three scripts and the retrieved content, write a script to perform multimodal fusion.                                                                                         |
| 30                                                          | <b>code</b>     | Based on the previous content, organically integrates all the code snippets into one comprehensive script that trains a multimodal HAR model using the training data provided by the user. |
| 31                                                          | <b>test</b>     | First, generate some test cases that input a small batch of the multimodal data into the HAR model to test whether the script can be executed.                                             |
| Several iterations of debugging is conducted.               |                 |                                                                                                                                                                                            |
| 35                                                          | <b>test</b>     | Now, execute the script by feeding the entire multimodal dataset into the model to train the model.                                                                                        |
| 36                                                          | <b>reason</b>   | Based on the evaluation results, I need to adopt more advanced methods for model training.                                                                                                 |
| 37                                                          | <b>retrieve</b> | Search for some SOTA but more advanced methods for extracting features from Wi-Fi data.                                                                                                    |
| Several iterations of debugging is conducted.               |                 |                                                                                                                                                                                            |
| 49                                                          | <b>reason</b>   | Finally, since the trained HAR model satisfies the resource constraints of the target device, the next step is to deploy it on the target device.                                          |
| 50                                                          | <b>code</b>     | Based on the user-provided information about how to connect to the device and transfer the script and model, write a script to implement that.                                             |
| 51                                                          | <b>test</b>     | Test the functionality correctness of the script.                                                                                                                                          |
| Several iterations of model optimization and deployment.    |                 |                                                                                                                                                                                            |
| 63                                                          | <b>reason</b>   | Based on the evaluation results, all requirements are fulfilled, the executable edge AI program is ...                                                                                     |

### 9.3.3 SE<sup>2</sup> Workflow for VLU

The agentic workflow explored by SE<sup>2</sup> for VLU is presented in the table below. Initially, SE<sup>2</sup> does not realize that the target device for deployment has a different runtime environment (*i.e.*, ARM architecture) and the generated script cannot be successfully executed. However, our *mobile runtime feedback mechanism* records such failures and returns the relevant runtime semantics to the *coordinator*. After carefully analyzing the runtime semantics, SE<sup>2</sup> realizes the runtime incompatibility and promptly modifies the script by first converting the model into a compatible ONNX format and then adopting the TensorRT engine for inference, which is compatible with Jetson Nano. *This further highlights the adaptability of SE<sup>2</sup> in exploring unknown cases with agile adjustment.*

| No.                                                                       | Operation       | Instruction                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------------------------------------------------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                                                                         | <b>retrieve</b> | Search for vision-language understanding.                                                                                                                                                                                                                                                                                                       |
| Several iterations of model optimization and deployment (similar to HAR). |                 |                                                                                                                                                                                                                                                                                                                                                 |
| 12                                                                        | <b>code</b>     | Based on the retrieved knowledge, write a script to build two embedding models for image and text processing, respectively.                                                                                                                                                                                                                     |
| Several iterations of debugging is conducted.                             |                 |                                                                                                                                                                                                                                                                                                                                                 |
| 37                                                                        | <b>code</b>     | Deploy the trained model on Jetson Nano.                                                                                                                                                                                                                                                                                                        |
| 38                                                                        | <b>test</b>     | Execute the code and record its performance.                                                                                                                                                                                                                                                                                                    |
| 39                                                                        | <b>reason</b>   | The script fails to execute, I need to analyze the runtime semantics fed back by the probe program. First, the script can be successfully executed on the PC but fails on Jetson Nano. The feedback log shows “incompatible runtime environments”, indicating that the model trained on PC requires a different runtime on Jetson Nano.         |
| 40                                                                        | <b>retrieve</b> | Search the runtime environment of Jetson Nano.                                                                                                                                                                                                                                                                                                  |
| 41                                                                        | <b>reason</b>   | Based on the retrieved content and runtime semantics, Jetson Nano is based on the ARM architecture, while the PC is based on the x86 architecture. They have different runtime environments and distinct versions of PyTorch with diverse features. Therefore, it is infeasible to directly employ the PC-trained PyTorch model on Jetson Nano. |
| 42                                                                        | <b>retrieve</b> | Search how to deploy AI models on Jetson Nano.                                                                                                                                                                                                                                                                                                  |
| 43                                                                        | <b>retrieve</b> | Search how to convert models into ONNX format and use TensorRT to perform inference tasks.                                                                                                                                                                                                                                                      |
| 52                                                                        | <b>code</b>     | Write a script to convert the trained models into ONNX format and use TensorRT for inference.                                                                                                                                                                                                                                                   |
| 53                                                                        | <b>test</b>     | Test the functionality correctness and performance.                                                                                                                                                                                                                                                                                             |
| 54                                                                        | <b>reason</b>   | Now the converted model can be successfully deployed on Jetson Nano to perform inference tasks. The next step is to evaluate its performance, including resource consumptions, inference latency, <i>etc.</i>                                                                                                                                   |
| 55                                                                        | <b>test</b>     | I need to evaluate the model using the test data residing on Jetson Nano and record various performance metrics.                                                                                                                                                                                                                                |
| Several iterations of model optimization and deployment.                  |                 |                                                                                                                                                                                                                                                                                                                                                 |

### 9.3.4 SE<sup>2</sup> Workflow for ODAD

The following table presents the agentic workflow constructed by SE<sup>2</sup>. To simulate real-world autonomous driving scenarios, we randomly remove some data modalities from the dataset. We can see that initially, the created model only accepts input data with fixed sizes. However, during on-device inference, the model fails because the shape of the input data sometimes mismatches with the expected size. SE<sup>2</sup> then add a few more steps to design an elastic AI model that accepts dynamic data input. *This demonstrates the superior adaptability of SE<sup>2</sup> in handling unforeseen real-world cases.*

| No.                                                      | Operation       | Instruction                                                                                                                                                                                                                                                                                                                                                |
|----------------------------------------------------------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1                                                        | <b>retrieve</b> | Search object detection in autonomous driving.                                                                                                                                                                                                                                                                                                             |
| The intermediate processes are similar and thus omitted. |                 |                                                                                                                                                                                                                                                                                                                                                            |
| 41                                                       | <b>code</b>     | Write a script to deploy the trained object detection model on the target device.                                                                                                                                                                                                                                                                          |
| 42                                                       | <b>test</b>     | Execute the script and monitor the model's real-time performance.                                                                                                                                                                                                                                                                                          |
| 43                                                       | <b>reason</b>   | The script fails due to input data size mismatch. It seems that some of the data may not strictly follow consistent sizes. Specifically, after carefully analyzing the failed case, it seems that sometimes one or some modalities may be missing due to the unstable performance of the sensors.                                                          |
| 44                                                       | <b>reason</b>   | Based on the current error information, I need to redesign the model to support input data with varying sizes.                                                                                                                                                                                                                                             |
| 45                                                       | <b>retrieve</b> | Search how to design elastic AI models that support input data with different sizes                                                                                                                                                                                                                                                                        |
| 46                                                       | <b>retrieve</b> | Search how to build elastic multimodal models that accept modality missing for object detection.                                                                                                                                                                                                                                                           |
| 47                                                       | <b>reason</b>   | Based on the retrieved content, I can try modality-specific encoders with masking, where a mask vector is introduced to indicate which modalities are present in the current data batch. Then, during feature fusion, instead of concatenation, only the available features are aggregated ( <i>e.g.</i> , via masked attention, gated sum, or averaging). |
| 48                                                       | <b>code</b>     | Based on the reasoning results, write a script to implement modality-specific encoders with masking.                                                                                                                                                                                                                                                       |
| 49                                                       | <b>test</b>     | Test the functionality correctness of the script.                                                                                                                                                                                                                                                                                                          |
| Several iterations of model optimization and deployment. |                 |                                                                                                                                                                                                                                                                                                                                                            |

## 9.4 Generated Code Comparison

### 9.4.1 Sample Code on HAR

Fig. 9.8 shows two sample code snippets generated by GitHub Copilot and  $\text{SE}^2$  for the multimodal HAR application, respectively. We can see that the code generated by GitHub Copilot directly applies AI models to the raw Wi-Fi data without pre-processing. On the contrary,  $\text{SE}^2$  first extracts Doppler features from the Wi-Fi data and then applies AI models for further processing. Since the Doppler features typically contain abundant Wi-Fi signal characteristics influenced by human activities, the program synthesized by  $\text{SE}^2$  can effectively extract useful features and achieve a higher HAR classification accuracy.

```

import numpy as np
import torch

class HAR_Multimodal_Model(nn.Module):
    def __init__(self, wifi_shape, rfid_shape, mmwave_shape):
        super().__init__()
        # AI Model Design
        .....

    def load_data(dataset_path):
        wifi = np.load('WIFI_data.npy')
        .....

model = HAR_Multimodal_Model(wifi_shape, rfid_shape,
mmwave_shape, num_classes).to(device)
.....
outputs = model(wifi, rfid, mmwave)

```

Directly feeding raw Wi-Fi data into the model

(a) Code generated by GitHub Copilot

```

def extract_doppler_features(wifi_data):
    .....
    # Spectral centroid and bandwidth
    freq_idx = np.arange(wifi_data.shape[2])
    centroid = np.sum(wifi_data * freq_idx, axis=2) /
(np.sum(wifi_data, axis=2) + 1e-8)
    bandwidth = np.sqrt(np.sum(((freq_idx[None, None, :]
- centroid[:, :, None]) ** 2) * wifi_data, axis=2) /
(np.sum(wifi_data, axis=2) + 1e-8))
    # Flatten all features
    .....
    return features

wifi = extract_doppler_features(wifi)
model = HAR_Multimodal_Model(wifi_shape, rfid_shape,
mmwave_shape, num_classes).to(device)
outputs = model(wifi, rfid, mmwave)

```

First extract Doppler features from raw Wi-Fi data and then apply neural networks

(b) Code generated by SE<sup>2</sup>

Figure 9.8: Sample code comparison on the multimodal HAR application.

### 9.4.2 Sample Code on VLU

Fig. 9.9 shows two sample codes generated by GitHub Copilot and  $SE^2$ , respectively. We can see that GitHub Copilot attempts to directly deploy the VLU model on Jetson Nano. However, as the Jetson Nano adopts the ARM architecture, whereas the VLU model is trained on an x86-based PC with a completely different framework, the model cannot be directly executed on Jetson Nano. In contrast,  $SE^2$  first converts the trained model into a cross-platform ONNX format and then leverages the TensorRT engine for inference, which is fully compatible with Jetson Nano.  *$SE^2$  can flexibly adjust its workflow by dynamically analyzing the current state and adding extra operations for further processing.*

```

.....
train(model, train_loader, val_loader, device)
print("Training complete. The model is saved to model.pth")
# Inference on Jetson Nano using the test set
.....
for image, text, labels in loader:
    for i in range(audio.size(0)):
        start = time.time()
        out = model(image[i:i+1], text[i:i+1])
        _, pred = torch.max(out, 1)
        correct += (pred == labels[i:i+1]).sum().item()
        total += 1
        latencies.append((time.time() - start) * 1000) # ms
acc = correct / total
avg_latency = np.mean(latencies)
max_ram = measure_ram()
print(f"Test Accuracy: {acc*100:.2f}%")

```

Directly deploy the trained model on Jetson Nano

(a) Code generated by GitHub Copilot

```

.....
torch.onnx.export(
    model, (dummy_image, dummy_text), onnx_path,
    input_names=['image', 'text'], output_names=['output'],
    opset_version=11, dynamic_axes={'image': {0: 'batch'},
    'text': {0: 'batch'}, 'output': {0: 'batch'}}
)
.....
# TensorRT inference using ONNX
def trt_inference(onnx_path, test_loader, audio_shape,
image_shape, radar_shape, num_classes, device):
    import tensorrt as trt
    import pycuda.driver as cuda
    import pycuda.autoinit
    TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
    with open(onnx_path, 'rb') as f:
        onnx_model = f.read()
    builder = trt.Builder(TRT_LOGGER)

```

First convert the trained model into ONNX format

Then, use TensorRT for inference

(b) Code generated by SE<sup>2</sup>

Figure 9.9: Sample code comparison on the multimodal VLU application.

### 9.4.3 Sample Code on ODAD

For ODAD, Fig. 9.10 presents two sample codes generated by GitHub Copilot and  $\text{SE}^2$ , respectively. To simulate the modality missing issue in real-world autonomous driving scenarios, we randomly remove some data modalities. GitHub Copilot does not realize such cases and simply creates an AI model that only accepts input data with fixed sizes. As a result, when encountering input data batches with missing modalities, GitHub Copilot repeatedly modifies the failed model's structures without recognizing the underlying problem, unless users provide further instructions. On the contrary,  $\text{SE}^2$  discovers the problem by analyzing the failed cases, including input data shape, intermediate outputs, and model structures. Accordingly,  $\text{SE}^2$  creates a *modality-specific encoders with masking* strategy to substitute the original fusion module that simply concatenates all features into one vector. In the proposed strategy, a mask vector is introduced to represent which modalities are absent in the current data batch, and only the available features are aggregated (*e.g.*, via masked attention, gated sum, or averaging).

```

class ODADModel(nn.Module):
    def __init__(self, camera_shape, radar_shape,
lidar_shape, gps_shape, num_classes):
        super().__init__()
        # Modality encoders
        self.camera_branch = .....
        self.radar_branch = .....
        self.lidar_branch = .....
        self.gps_branch = .....
        # Fusion and output
        self.fusion = .....

    def forward(self, camera, radar, lidar, gps):
        camera = self.camera_branch(camera)
        radar = self.radar_branch(radar)
        lidar = self.lidar_branch(lidar)
        gps = self.gps_branch(gps)
        x = torch.cat([camera, radar, lidar, gps], dim=1)
        out = self.fusion(x)
        return out

```

Only support input data with fixed sizes

Crushes when some modalities are missing

(a) Code generated by GitHub Copilot

```

class ODADModel(nn.Module):
    def __init__(self, shape_list, num_classes):
        .....
    def forward(self, camera, radar, lidar, gps, mask=None):
        feats = []
        if mask[0] is not None:
            feats.append(self.camera_branch(camera))
        if mask[1] is not None:
            feats.append(self.radar_branch(camera))
        if lidar is not None: .....
        if gps is not None: .....

        masked_feature = None
        for modality_index in mask:
            masked_feature += feats[modality_index]
        sum_feats = masked_feats.sum(dim=1)
        denom = mask.sum(dim=1).clamp(min=1)
        agg_feat = sum_feats / denom
        return self.fusion(agg_feat)

```

Modality-specific mask

Feature aggregation

(b) Code generated by SE<sup>2</sup>

Figure 9.10: Sample code comparison on the multimodal ODAD application.

## 9.5 Agentic Recovery Mechanism

In  $\text{SE}^2$ , early-stage errors may lead to accumulated downstream errors, manifesting in two aspects: 1) *Runtime errors* occur when agents generate malformed outputs that cannot be parsed by subsequent agents, or when tools raise exceptions. They can be handled by existing works using formal language [302], which are orthogonally integrated into  $\text{SE}^2$ . 2) *Semantic misalignment* occurs when the output is logically incorrect or misaligned with intended tasks. Such errors are more difficult to detect as they do not trigger execution errors but lead to incorrect reasoning, flawed code logic, *etc.*

**Challenge & Solution.** To handle semantic misalignment, existing works often employ extra agents guided by handcrafted prompts for error monitoring. For instance, AutoIOT [4] defines numerous rules to iteratively refine the code. Developers need to design specific detection rules at each step based on agents’ predefined tasks. In contrast,  $\text{SE}^2$  faces significant challenges in designing a one-size-fits-all semantic misalignment monitor, as both workflows and the tasks at each step are dynamically determined. To tackle this, our insight is that the *coordinator* can serve as a supervisor, overseeing each atomic agent’s activities and providing suggestions for regeneration if errors occur. To this end, we propose a *semantic alignment grading* to criticize atomic agents.

**Semantic Alignment Grading.** As mentioned in § 5.4.2, we consider two types of semantic misalignment: local and global alignment. We prompt the *coordinator* to assign two scores to each response generated by atomic agents based on six designed grading levels: 1) *Completely off-track*. The response contains critical failures and the content is incorrect or disrupts the workflow. 2) *Misaligned and almost useless*. The response indicates semantic drift or wrong tool use to follow instructions. 3) *Poorly aligned but slightly useful*. The agent fails to complete the instruction with a little

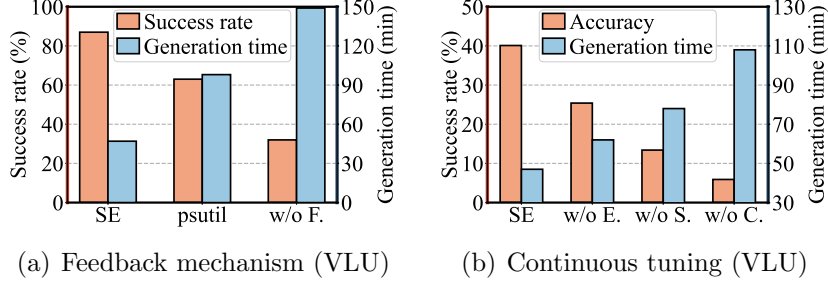


Figure 9.11: Ablation study on VLU.

related content. 4) *Partially aligned with noticeable issues*. The response indicates a meaningful deviation (*e.g.*, incomplete reasoning, vague retrieval, general algorithms not tailored to AIoT). 5) *Mostly aligned with minor flaws*. The response captures slight deviations that don't require correction. 6) *Perfectly aligned*. The response has no ambiguity and is well-aligned with the instruction or the user requirement.

The graded scores will be integrated into *coordinator*'s feedback for further fine-tuning (§ 5.4.2 & § 5.4.3).

## 9.6 More Ablation Studies

Fig. 9.11 and Fig. 9.12 show the results of our ablation study on VLU and ODAD, respectively. We see that the results are similar to those in HAR. Notably, the performance gap after removing the feedback mechanism and the *continuous tuning paradigm* enlarges when the edge AI tasks become more complex. Specifically, as shown in Fig. 9.11(a) and Fig. 9.11(b), the success rate drops dramatically when these two modules are removed. This is because in VLU evaluations, we request  $\text{SE}^2$  to deploy the model on a Jetson Nano board, whose runtime environment is different from that of the PC where  $\text{SE}^2$  resides. Thus, without the runtime semantics as feedback, the success rate of executable program synthesis decreases, and  $\text{SE}^2$  requires more agentic steps to discover this issue with a longer generation time. Similar

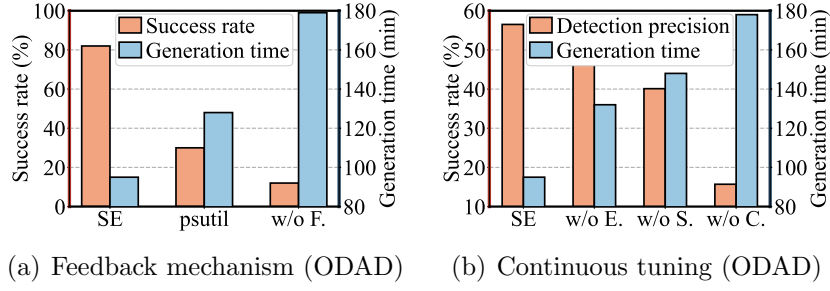


Figure 9.12: Ablation study on ODAD.

results can also be observed in ODAD evaluations (Fig. 9.12(a) and Fig. 9.12(b)), which involve unexpected modality missing issues.

# References

- [1] M. A. Islam, M. E. Ali, and M. R. Parvez, “Mapcoder: Multi-agent code generation for competitive problem solving,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 4912–4944.
- [2] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, “Agentcoder: Multi-agent-based code generation with iterative testing and optimisation,” *arXiv preprint arXiv:2312.13010*, 2023.
- [3] Y. Shang, Y. Li, K. Zhao, L. Ma, J. Liu, F. Xu, and Y. Li, “Agentsquare: Automatic llm agent search in modular design space,” *arXiv preprint arXiv:2410.06153*, 2024.
- [4] L. Shen, Q. Yang, Y. Zheng, and M. Li, “Autoiot: Llm-driven automated natural language programming for aiot applications,” in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 468–482.
- [5] L. Shen, Q. Yang, X. Huang, Z. Ma, and Y. Zheng, “Gpiot: Tailoring small language models for iot program synthesis and development,” in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 199–212.
- [6] K. Gong, W. Dong, H. Wang, Y. Peng, and Y. Gao, “Programming embedded iot applications in natural language with iotpilot,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 70–82.
- [7] H. Yang, M. Li, M. Han, Z. Li, and W. Xu, “Embedgenius: Towards automated software development for generic embedded iot systems,” *arXiv preprint arXiv:2412.09058*, 2024.

- [8] S. I. Siam, H. Ahn, L. Liu, S. Alam, H. Shen, Z. Cao, N. Shroff, B. Krishnamachari, M. Srivastava, and M. Zhang, “Artificial intelligence of things: A survey,” *ACM Transactions on Sensor Networks*, vol. 21, no. 1, pp. 1–75, 2025.
- [9] Q. Yang and Y. Zheng, “Neural enhanced underwater sos detection,” in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 971–980.
- [10] Q. Yang, J. Stuchbury-Wass, T. A. Bytheway, and C. Mascolo, “Openbrush: An open-source manual toothbrush for force and motion sensing,” in *Companion of the 2025 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2025, pp. 1297–1300.
- [11] Q. Yang, “Ubiquitous acoustic intelligence: Toward intuitive, resilient, and secure mobile sensing systems,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 655–656.
- [12] Q. Yang and Y. Zheng, “Aquahelper: Underwater sos transmission and detection in swimming pools,” in *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, 2023, pp. 294–307.
- [13] Q. Yang, Y. Liu, J. Stuchbury-Wass, K.-J. Butkow, E. Panariti, D. Ma, and C. Mascolo, “Smarteeth: Augmenting manual toothbrushing with in-ear microphones,” in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–19.
- [14] Q. Yang, Y. Liu, J. Stuchbury-Wass, K.-J. Butkow, D. Ma, and C. Mascolo, “Brushbuds: Toothbrushing tracking using earphone imus,” in *Companion of the 2024 on ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2024, pp. 655–660.
- [15] Q. Yang and Y. Zheng, “Model-based head orientation estimation for smart devices,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 5, no. 3, pp. 1–24, 2021.
- [16] Q. Yang, K. Cui, and Y. Zheng, “Room-scale voice liveness detection for smart devices,” *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 5, pp. 4982–4996, 2024.
- [17] K. Cui, Q. Yang, L. Shen, Y. Zheng, F. Xiao, and J. Han, “Towards isac-empowered mmwave radars by capturing modulated vibrations,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 13 787–13 803, 2024.

- [18] K. Cui, L. Shen, Y. Zheng, F. Xiao, and J. Han, “Talk2radar: Talking to mmwave radars via smartphone speaker,” in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 2358–2367.
- [19] K. Cui, Q. Yang, Y. Zheng, and J. Han, “mmripple: Communicating with mmwave radars through smartphone vibration,” in *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, 2023, pp. 149–162.
- [20] K. Cui, Q. Yang, L. Shen, Y. Zheng, and J. Han, “Integrated sensing and communication between daily devices and mmwave radars,” in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 831–832.
- [21] K. Cui, Y. Wang, Y. Zheng, and J. Han, “Shakereader: readuhf rfid using smartphone,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 3, pp. 1793–1809, 2021.
- [22] Arduino, “Arduino - Home,” <https://www.arduino.cc/>, 2026, open-source electronics platform.
- [23] Espressif Systems, “ESP32 Wi-Fi & Bluetooth MCU,” <https://www.espressif.com/en/products/socs/esp32>, 2016, system on a Chip (SoC).
- [24] NVIDIA Corporation, “NVIDIA Jetson Nano Developer Kit,” <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>, 2019, edge AI hardware platform.
- [25] Raspberry Pi Ltd, “Raspberry Pi 5,” <https://www.raspberrypi.com/products/raspberry-pi-5/>, 2023, single-board computer.
- [26] L. Shen, Q. Yang, K. Cui, Y. Zheng, X.-Y. Wei, J. Liu, and J. Han, “Fed-conv: A learning-on-model paradigm for heterogeneous federated clients,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 398–411.
- [27] L. Shen and Y. Zheng, “Feddm: data and model heterogeneity-aware federated learning via dynamic weight sharing,” in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 975–976.
- [28] Q. Yang, K. Cui, and Y. Zheng, “Voshield: Voice liveness detection with sound field dynamics,” in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 2023, pp. 1–10.

- [29] Q. Yang and Y. Zheng, “Deepear: Sound localization with binaural microphones,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 1, pp. 359–375, 2022.
- [30] M. Gao, L. Zhang, L. Shen, X. Zou, J. Han, F. Lin, and K. Ren, “Kite: Exploring the practical threat from acoustic transduction attacks on inertial sensors,” in *Proceedings of the 20th ACM conference on embedded networked sensor systems*, 2022, pp. 696–709.
- [31] J. Liu, W. Song, L. Shen, J. Han, and K. Ren, “Secure user verification and continuous authentication via earphone imu,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 11, pp. 6755–6769, 2022.
- [32] Q. Xue, X. Yin, B. Yang, and W. Gao, “Phyt2v: Llm-guided iterative self-refinement for physics-grounded text-to-video generation,” in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025, pp. 18 826–18 836.
- [33] J. Zhang, P. Li, F. Li, D. Zhang, W. Gao, S. Yue, Y. Zhang, and J. Ren, “Focusx: All-in-focus image synthesis for dynamic scenes on mobile devices,” in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 667–681.
- [34] W. Yin, D. Xu, G. Huang, Y. Zhang, S. Wei, M. Xu, and X. Liu, “Piebridge: Fast and parameter-efficient on-device training via proxy networks,” in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, 2024, pp. 126–140.
- [35] W. Yin, R. Yi, D. Xu, G. Huang, M. Xu, and X. Liu, “Elastic on-device llm service,” in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 984–999.
- [36] S. Joel, J. Wu, and F. Fard, “A survey on llm-based code generation for low-resource and domain-specific programming languages,” *ACM Transactions on Software Engineering and Methodology*, 2024.
- [37] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, “A survey on code generation with llm-based agents,” *arXiv preprint arXiv:2508.00083*, 2025.
- [38] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.

- [39] L. Shen and Y. Zheng, “Iotcoder: A copilot for iot application development,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 1647–1649.
- [40] S. Butterworth *et al.*, “On the theory of filter amplifiers,” *Wireless Engineer*, vol. 7, no. 6, pp. 536–541, 1930.
- [41] GitHub, “Github copilot,” <https://github.com/features/copilot>, 2026, aI code generation tool.
- [42] M. Babiuch and P. Smutný, “Benchmarking large language models for embedded systems programming in microcontroller-driven iot applications,” *Future Internet*, vol. 18, no. 2, p. 94, 2026.
- [43] S. Neel and P. Chang, “Privacy issues in large language models: A survey,” *arXiv preprint arXiv:2312.06717*, 2023.
- [44] E. Erdil, “Inference economics of language models,” *arXiv preprint arXiv:2506.04645*, 2025.
- [45] NVIDIA Corporation, “NVIDIA TensorRT,” <https://github.com/NVIDIA/TensorRT>, 2026, version 10.15.1.
- [46] Intel Corporation, “OpenVINO™ Toolkit,” <https://github.com/openvinotoolkit/openvino>, 2026.
- [47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “[TensorFlow]: a system for {Large-Scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
- [48] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [49] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [50] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.

- [51] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong *et al.*, “Chatdev: Communicative agents for software development,” in *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2024, pp. 15 174–15 186.
- [52] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, “Metagpt: Meta programming for a multi-agent collaborative framework,” in *The twelfth international conference on learning representations*, 2023.
- [53] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “Swe-agent: Agent-computer interfaces enable automated software engineering,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.
- [54] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, “Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 38 154–38 180, 2023.
- [55] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, “Autogen: Enabling next-gen llm applications via multi-agent conversations,” in *First conference on language modeling*, 2024.
- [56] F. Sarhaddi, N. T. Nguyen, A. Zuniga, M. Khan, P. Hui, S. Tarkoma, H. Flores, and P. Nurmi, “Llms and iot: A comprehensive survey on large language models and the internet of things,” *TechRxiv*, 2026.
- [57] I. Adabara, B. O. Sadiq, A. N. Shuaibu, Y. I. Danjuma, and V. Maninti, “Trustworthy agentic ai systems: a cross-layer review of architectures, threat models, and governance strategies for real-world deployment,” *F1000Research*, vol. 14, no. 905, p. 905, 2025.
- [58] N. Guainazzo, G. Delzanno, D. Ancona, and D. DAgostino, “Navigating the seas of ai: Effectiveness of small language models on edge devices for maritime applications,” *Sensors*, vol. 26, no. 5, p. 1590, 2026.
- [59] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, “Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment, 2023,” *arXiv preprint arXiv:2312.12148*, 2023.
- [60] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, “Lora: Low-rank adaptation of large language models.” *ICLR*, vol. 1, no. 2, p. 3, 2022.

- [61] S. Hu, C. Lu, and J. Clune, “Automated design of agentic systems,” *arXiv preprint arXiv:2408.08435*, 2024.
- [62] J. Zhang, J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, M. Zhuge, X. Cheng, S. Hong, J. Wang *et al.*, “Aflow: Automating agentic workflow generation,” *arXiv preprint arXiv:2410.10762*, 2024.
- [63] S. Lee, J. Choi, J. Lee, M. H. Wasi, H. Choi, S. Ko, S. Oh, and I. Shin, “Mobilegpt: Augmenting llm with human-like app memory for mobile task automation,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 1119–1133.
- [64] D. Cai, Y. Wu, S. Wang, F. X. Lin, and M. Xu, “Efficient federated learning for modern nlp,” in *Proceedings of the 29th annual international conference on mobile computing and networking*, 2023, pp. 1–16.
- [65] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, “Autodroid: Llm-powered task automation in android,” in *Proceedings of the 30th annual international conference on Mobile computing and networking*, 2024, pp. 543–557.
- [66] K. Huang and W. Gao, “Real-time neural network inference on extremely weak devices: agile offloading with explainable ai,” in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022, pp. 200–213.
- [67] J. Liu, W. Song, L. Shen, J. Han, X. Xu, and K. Ren, “Mandipass: Secure and usable user authentication via earphone imu,” in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 674–684.
- [68] X. Ouyang, X. Shuai, Y. Li, L. Pan, X. Zhang, H. Fu, S. Cheng, X. Wang, S. Cao, J. Xin *et al.*, “Admarker: A multi-modal federated learning system for monitoring digital biomarkers of alzheimer’s disease,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 404–419.
- [69] X. Ouyang, Z. Xie, H. Fu, S. Cheng, L. Pan, N. Ling, G. Xing, J. Zhou, and J. Huang, “Harmony: Heterogeneous multi-modal federated learning through disentangled model training,” in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, 2023, pp. 530–543.

- [70] Y. He, C. Bian, J. Xia, S. Shi, Z. Yan, Q. Song, and G. Xing, “Vi-map: Infrastructure-assisted real-time hd mapping for autonomous driving,” in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–15.
- [71] H. Xu, L. Han, Q. Yang, M. Li, and M. Srivastava, “Penetrative ai: Making llms comprehend the physical world,” in *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*, 2024, pp. 1–7.
- [72] X. Liu, D. McDuff, G. Kovacs, I. Galatzer-Levy, J. Sunshine, J. Zhan, M.-Z. Poh, S. Liao, P. Di Achille, and S. Patel, “Large language models are few-shot health learners,” *arXiv preprint arXiv:2305.15525*, 2023.
- [73] S. Ji, X. Zheng, and C. Wu, “Hargpt: Are llms zero-shot human activity recognizers?” in *2024 IEEE International Workshop on Foundation Models for Cyber-Physical Systems & Internet of Things (FMSys)*. IEEE, 2024, pp. 38–43.
- [74] P. Quan, X. Ouyang, J. V. Jeyakumar, Z. Wang, Y. Xing, and M. Srivastava, “Sensorbench: Benchmarking llms in coding-based sensor processing,” in *Proceedings of the 26th International Workshop on Mobile Computing Systems and Applications*, 2025, pp. 25–30.
- [75] S. Yu, X. Xia, Z. Zhang, N. Hou, and Y. Zheng, “Fdlora: Tackling downlink-uplink asymmetry with full-duplex lora gateways,” in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, 2024, pp. 281–294.
- [76] S. Yu, X. Xia, N. Hou, Y. Zheng, and T. Gu, “Revolutionizing lora gateway with xgate: Scalable concurrent transmission across massive logical channels,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 482–496.
- [77] S. Yu, Z. Zhang, X. Xia, Y. Zheng, and J. Wang, “Are lora logical channels really orthogonal? practically orthogonalizing massive logical channels,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 56–69.
- [78] N. Hou, X. Xia, Y. Wang, and Y. Zheng, “One shot for all: Quick and accurate data aggregation for lpwans,” *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 2285–2298, 2024.
- [79] N. Hou, X. Xia, and Y. Zheng, “Jamming of lora phy and countermeasure,” *ACM Transactions on Sensor Networks*, vol. 19, no. 4, pp. 1–27, 2023.

- [80] J. Cao, C. Lin, Y. Liu, and Z. Li, “Gaze tracking on any surface with your phone,” in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 320–333.
- [81] N. Hou, Y. Wang, X. Xia, S. Yu, Y. Zheng, and T. Gu, “Molara: Intelligent mobile antenna system for enhanced lora reception in urban environments,” in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 424–436.
- [82] Y. Yang, L. Jiao, and Y. Xu, “A queueing theoretic perspective on low-latency llm inference with variable token length,” in *2024 22nd International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*. IEEE, 2024, pp. 273–280.
- [83] H. Le, H. Chen, A. Saha, A. Gokul, D. Sahoo, and S. Joty, “Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules,” *arXiv preprint arXiv:2310.08992*, 2023.
- [84] S. Jiang, Y. Wang, and Y. Wang, “Selfevolve: A code evolution framework via large language models,” *arXiv preprint arXiv:2306.02907*, 2023.
- [85] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, “Wizardcoder: Empowering code large language models with evol-instruct,” *arXiv preprint arXiv:2306.08568*, 2023.
- [86] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [87] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [88] M. Du, A. T. Luu, B. Ji, and S.-K. Ng, “Mercury: An efficiency benchmark for llm code synthesis,” *arXiv preprint arXiv:2402.07844*, 2024.
- [89] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [90] R. Jain and U. Suman, “A systematic literature review on global software development life cycle,” *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 1–14, 2015.

- [91] H. Chase *et al.*, “Langchain: Build context-aware reasoning applications,” 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>
- [92] A. Dunn, J. Dagdelen, N. Walker, S. Lee, A. S. Rosen, G. Ceder, K. Persson, and A. Jain, “Structured information extraction from complex scientific text with fine-tuned large language models,” *arXiv preprint arXiv:2212.05238*, 2022.
- [93] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy *et al.*, “Text and code embeddings by contrastive pre-training,” *arXiv preprint arXiv:2201.10005*, 2022.
- [94] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, “The faiss library,” *IEEE Transactions on Big Data*, 2025.
- [95] Q. Wang, Y. Fu, Y. Cao, S. Wang, Z. Tian, and L. Ding, “Recursively summarizing enables long-term dialogue memory in large language models,” *Neurocomputing*, vol. 639, p. 130193, 2025.
- [96] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” *arXiv preprint arXiv:2304.05128*, 2023.
- [97] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegrefe, U. Alon, N. Dziri, S. Prabhume, Y. Yang *et al.*, “Self-refine: Iterative refinement with self-feedback,” *Advances in neural information processing systems*, vol. 36, pp. 46 534–46 594, 2023.
- [98] Tavily AI, “tavily-python,” 2022. [Online]. Available: <https://github.com/tavily-ai/tavily-python>
- [99] L. Wu, X. Xie, and Y. Wang, “Ecg enhancement and r-peak detection based on window variability,” in *Healthcare*, vol. 9, no. 2. MDPI, 2021, p. 227.
- [100] G. B. Moody and R. G. Mark, “The impact of the mit-bih arrhythmia database,” *IEEE engineering in medicine and biology magazine*, vol. 20, no. 3, pp. 45–50, 2001.
- [101] P. Hamilton, “Open source ecg analysis,” in *Computers in cardiology*. IEEE, 2002, pp. 101–104.
- [102] I. I. Christov, “Real time electrocardiogram qrs detection using combined adaptive threshold,” *Biomedical engineering online*, vol. 3, no. 1, p. 28, 2004.

- [103] W. A. Engelse and C. Zeelenberg, “A single scan algorithm for qrs-detection and feature extraction,” *Computers in cardiology*, vol. 6, no. 1979, pp. 37–42, 1979.
- [104] J. Pan and W. J. Tompkins, “A real-time qrs detection algorithm,” *IEEE transactions on biomedical engineering*, no. 3, pp. 230–236, 2007.
- [105] V. Kalidas and L. Tamil, “Real-time qrs detector using stationary wavelet transform for automated ecg analysis,” in *2017 IEEE 17th international conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 2017, pp. 457–461.
- [106] M. Gao, L. Zhang, L. Shen, X. Zou, J. Han, F. Lin, and K. Ren, “Exploring practical acoustic transduction attacks on inertial sensors in mdof systems,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 5, pp. 3539–3557, 2023.
- [107] H. Xu, P. Zhou, R. Tan, M. Li, and G. Shen, “Limu-bert: Unleashing the potential of unlabeled data for imu sensing applications,” in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 220–233.
- [108] <https://github.com/bartkowiaktomasz/har-wisdm-lstm-rnns>, 2018.
- [109] [https://github.com/akshaykulkarni07/activity\\_recognition](https://github.com/akshaykulkarni07/activity_recognition), 2018.
- [110] [https://github.com/coloriz/HAR-WISDM\\_ar](https://github.com/coloriz/HAR-WISDM_ar), 2020.
- [111] <https://github.com/bartkowiaktomasz/har-wisdm-bidirectional-lstm-rnns>, 2018.
- [112] <https://github.com/AthanJohn/Human-Activity-Recognition-WISDM>, 2022.
- [113] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, “Activity recognition using cell phone accelerometers,” *ACM SigKDD Explorations Newsletter*, vol. 12, no. 2, pp. 74–82, 2011.
- [114] H. Xue, Q. Cao, Y. Ju, H. Hu, H. Wang, A. Zhang, and L. Su, “M4esh: mmwave-based 3d human mesh construction for multiple subjects,” in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 391–406.

- [115] F. Wang, Y. Lv, M. Zhu, H. Ding, and J. Han, “Xrf55: A radio frequency dataset for human indoor action analysis,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 8, no. 1, pp. 1–34, 2024.
- [116] G. Rodolà, “psutil: Cross-platform lib for process and system monitoring in python,” 2026, gitHub repository. [Online]. Available: <https://github.com/giampaolo/psutil/releases/tag/release-X.Y.Z>
- [117] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [118] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.
- [119] T. Cohere, A. Ahmadian, M. Ahmed, J. Alammam, M. Alizadeh, Y. Alnumay, S. Althammer, A. Arkhangorodsky, V. Aryabumi, D. Aumiller *et al.*, “Command a: An enterprise-ready large language model,” *arXiv preprint arXiv:2504.00698*, 2025.
- [120] Anthropic, “Claude 2,” <https://www.anthropic.com/news/claude-2>, Jul. 2023, accessed: 2026-04-19.
- [121] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love *et al.*, “Gemma: Open models based on gemini research and technology,” *arXiv preprint arXiv:2403.08295*, 2024.
- [122] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, “Lost in the middle: How language models use long contexts,” *Transactions of the association for computational linguistics*, vol. 12, pp. 157–173, 2024.
- [123] 2024. [Online]. Available: <https://www.sonarsource.com/>
- [124] 2024. [Online]. Available: <https://rules.sonarsource.com/python/RSPEC-2316/>
- [125] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, “Deep learning based code smell detection,” *IEEE transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2019.

- [126] A. Tamkin, M. Brundage, J. Clark, and D. Ganguli, “Understanding the capabilities, limitations, and societal impact of large language models,” *arXiv preprint arXiv:2102.02503*, 2021.
- [127] L. Floridi and M. Chiriatti, “Gpt-3: Its nature, scope, limits, and consequences,” *Minds and machines*, vol. 30, no. 4, pp. 681–694, 2020.
- [128] K. Wang, Z. Zhou, and Z. Li, “Latte: Layer algorithm-aware training time estimation for heterogeneous federated learning,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 1470–1484.
- [129] J.-T. Chen and C.-M. Huang, “Forgetful large language models: Lessons learned from using llms in robot programming,” in *Proceedings of the AAAI Symposium Series*, vol. 2, no. 1, 2023, pp. 508–513.
- [130] K. Shi, X. Sun, Q. Li, and G. Xu, “Compressing long context for enhancing rag with amr-based concept distillation,” *arXiv preprint arXiv:2405.03085*, 2024.
- [131] S. Sun, R. Yuan, Z. Cao, W. Li, and P. Liu, “Prompt chaining or stepwise prompt? refinement in text summarization,” in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 7551–7558.
- [132] X. Ouyang and M. Srivastava, “Llmsense: Harnessing llms for high-level reasoning over spatiotemporal sensor traces,” *arXiv preprint arXiv:2403.19857*, 2024.
- [133] C. Chang, W.-Y. Wang, W.-C. Peng, and T.-F. Chen, “Llm4ts: Aligning pre-trained llms as data-efficient time-series forecasters,” *ACM Transactions on Intelligent Systems and Technology*, vol. 16, no. 3, pp. 1–20, 2025.
- [134] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, “Deepseek-coder: when the large language model meets programming—the rise of code intelligence,” *arXiv preprint arXiv:2401.14196*, 2024.
- [135] L. Zhong, Z. Wang, and J. Shang, “Debug like a human: A large language model debugger via verifying runtime execution step by step,” in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 851–870.
- [136] S. Duan, N. Kanakaris, X. Xiao, H. Ping, C. Zhou, N. K. Ahmed, G. Ma, M. Capota, T. L. Willke, S. Nazarian *et al.*, “Leveraging reinforcement learning

- and large language models for code optimization,” *arXiv e-prints*, pp. arXiv-2312, 2023.
- [137] Significant Gravitass, “Autogpt.” [Online]. Available: <https://agpt.co>
- [138] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang *et al.*, “Efficient large language models: A survey,” *arXiv preprint arXiv:2312.03863*, 2023.
- [139] X. Wang, Y. Zheng, Z. Wan, and M. Zhang, “Svd-llm: Truncation-aware singular value decomposition for large language model compression,” *arXiv preprint arXiv:2403.07378*, 2024.
- [140] J. Nie, H. Shao, Y. Fan, Q. Shao, H. You, M. Preindl, and X. Jiang, “Llm-based conversational ai therapist for daily functioning screening and psychotherapeutic intervention via everyday smart devices,” *arXiv preprint arXiv:2403.10779*, 2024.
- [141] K. Liu, B. Yang, L. Xu, Y. Guo, N. Ling, Z. Zhao, G. Xing, X. Shuai, X. Ren, X. Jiang *et al.*, “Tasking heterogeneous sensor systems with llms,” in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, 2024, pp. 901–902.
- [142] T. An, Y. Zhou, H. Zou, and J. Yang, “Iot-llm: A framework for enhancing large language model reasoning from real-world sensor data,” *Patterns*, vol. 7, no. 1, 2026.
- [143] J. Cao, J. Chen, C. Lin, Y. Liu, K. Wang, and Z. Li, “Practical gaze tracking on any surface with your phone,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 14 689–14 707, 2024.
- [144] N. Hou, X. Xia, and Y. Zheng, “Dont miss weak packets: Boosting lora reception with antenna diversities,” *ACM Transactions on Sensor Networks*, vol. 19, no. 2, pp. 1–25, 2023.
- [145] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 35, no. 2, pp. 1–72, 2026.
- [146] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, pp. 1–124, 2023.

- [147] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
- [148] H. Ding, Z. Fan, I. Guehring, G. Gupta, W. Ha, J. Huan, L. Liu, B. Omidvar-Tehrani, S. Wang, and H. Zhou, “Reasoning and planning with large language models in code development,” in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6480–6490.
- [149] G. Xiong, Q. Jin, X. Wang, M. Zhang, Z. Lu, and A. Zhang, “Improving retrieval-augmented generation in medicine with iterative follow-up questions,” in *Biocomputing 2025: Proceedings of the Pacific Symposium*. World Scientific, 2024, pp. 199–214.
- [150] K. Wu, E. Wu, and J. Zou, “How faithful are rag models? quantifying the tug-of-war between rag and llms internal prior,” *arXiv preprint arXiv:2404.10198*, vol. 3, no. 1, 2024.
- [151] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, “Parrot: Efficient serving of {LLM-based} applications with semantic variable,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 929–945.
- [152] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, “Improving llm code generation with grammar augmentation,” *arXiv preprint arXiv:2403.01632*, vol. 19, 2024.
- [153] IBM, “What are small language models?” 2024. [Online]. Available: <https://www.ibm.com/think/topics/small-language-models>
- [154] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.
- [155] Z. Lu, X. Li, D. Cai, R. Yi, F. Liu, X. Zhang, N. D. Lane, and M. Xu, “Small language models: Survey, measurements, and insights,” *arXiv preprint arXiv:2409.15790*, 2024.
- [156] S. Wu, H. Fei, L. Qu, W. Ji, and T.-S. Chua, “Next-gpt: Any-to-any multimodal llm,” in *Forty-first International Conference on Machine Learning*, 2024.

- [157] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, “Wizardlm: Empowering large language models to follow complex instructions,” *arXiv preprint arXiv:2304.12244*, 2023.
- [158] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
- [159] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, “Calibrate before use: Improving few-shot performance of language models,” in *International conference on machine learning*. Pmlr, 2021, pp. 12 697–12 706.
- [160] S. Chen, S. Wong, L. Chen, and Y. Tian, “Extending context window of large language models via positional interpolation,” *arXiv preprint arXiv:2306.15595*, 2023.
- [161] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, “Scipy 1.0: fundamental algorithms for scientific computing in python,” *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.
- [162] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [163] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “Qlora: Efficient finetuning of quantized llms,” *Advances in neural information processing systems*, vol. 36, pp. 10 088–10 115, 2023.
- [164] N. Ding, X. Lv, Q. Wang, Y. Chen, B. Zhou, Z. Liu, and M. Sun, “Sparse low-rank adaptation of pre-trained language models,” in *Proceedings of the 2023 conference on empirical methods in natural language processing*, 2023, pp. 4133–4145.
- [165] N. Shazeer, “Glu variants improve transformer,” *arXiv preprint arXiv:2002.05202*, 2020.
- [166] B. Muthukadan *et al.*, “Selenium with python.” [Online]. Available: <https://github.com/baijum/selenium-python>
- [167] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj, “Training large neural networks with constant memory using a new execution algorithm,” *arXiv preprint arXiv:2002.05645*, 2020.

- [168] S. Ji, Y. Xie, and M. Li, “Sifall: Practical online fall detection with rf sensing,” in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 2022, pp. 563–577.
- [169] J. Cao, Y. Liu, L. Han, and Z. Li, “Finger tracking using wrist-worn emg sensors,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 14 099–14 110, 2024.
- [170] S. Ji, X. Zhang, Y. Zheng, and M. Li, “Construct 3d hand skeleton with commercial wifi,” in *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, 2023, pp. 322–334.
- [171] A. Adhikari and S. Sur, “Misleep: Human sleep posture identification from deep learning augmented millimeter-wave wireless systems,” *ACM Transactions on Internet of Things*, vol. 5, no. 2, pp. 1–33, 2024.
- [172] L. Guo, L. Wang, C. Lin, J. Liu, B. Lu, J. Fang, Z. Liu, Z. Shan, J. Yang, and S. Guo, “Wiar: A public dataset for wifi-based activity recognition,” *IEEE Access*, vol. 7, pp. 154 935–154 945, 2019.
- [173] N. Ling, K. Wang, Y. He, G. Xing, and D. Xie, “Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms,” in *Proceedings of the 19th ACM conference on embedded networked sensor systems*, 2021, pp. 1–14.
- [174] J. Fan, M. Qi, L. Liu, and H. Ma, “Diffusion-driven incomplete multimodal learning for air quality prediction,” *ACM Transactions on Internet of Things*, vol. 6, no. 1, pp. 1–24, 2025.
- [175] T. Wang, Z. Li, H. Yan, X. Liu, B. Liu, S. Li, Z. Ma, J. Hu, D. Zhang, and T. Gu, “Audioguard: omnidirectional indoor intrusion detection using audio device,” *ACM Transactions on Internet of Things*, vol. 5, no. 1, pp. 1–22, 2023.
- [176] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, “Qwen technical report,” *arXiv preprint arXiv:2309.16609*, 2023.
- [177] J. Tejedor, C. A. García, D. G. Márquez, R. Raya, and A. Otero, “Multiple physiological signals fusion techniques for improving heartbeat detection: A review,” *Sensors*, vol. 19, no. 21, p. 4708, 2019.
- [178] D. Han, S. K. Bashar, J. Lázaro, F. Mohagheghian, A. Peitzsch, N. Nishita, E. Ding, E. L. Dickson, D. DiMezza, J. Scott *et al.*, “A real-time ppg peak

detection method for accurate determination of heart rate during sinus rhythm and cardiac arrhythmia,” *Biosensors*, vol. 12, no. 2, p. 82, 2022.

- [179] W. Wang, A. X. Liu, M. Shahzad, K. Ling, and S. Lu, “Understanding and modeling of wifi signal based human activity recognition,” in *Proceedings of the 21st annual international conference on mobile computing and networking*, 2015, pp. 65–76.
- [180] H. Zhou, T. Lu, Y. Liu, S. Zhang, and M. Gowda, “Learning on the rings: Self-supervised 3d finger motion tracking using wearable sensors,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 6, no. 2, pp. 1–31, 2022.
- [181] J. Choi, H. Y. Yeom, and Y. Kim, “Implementing cuda unified memory in the pytorch framework,” in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE, 2021, pp. 20–25.
- [182] C. Lin, K. Wang, Z. Li, and Y. Pu, “A workload-aware dvfs robust to concurrent tasks for mobile devices,” in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–16.
- [183] L. Shen, Q. Yang, K. Cui, Y. Zheng, X.-Y. Wei, J. Liu, and J. Han, “Hierarchical and heterogeneous federated learning via a learning-on-model paradigm,” *IEEE Transactions on Mobile Computing*, 2025.
- [184] K. Wang, J. Cao, Z. Zhou, and Z. Li, “Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 9, pp. 8935–8950, 2024.
- [185] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, “Codet5+: Open code large language models for code understanding and generation,” in *Proceedings of the 2023 conference on empirical methods in natural language processing*, 2023, pp. 1069–1088.
- [186] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, “{ServerlessLLM}:{Low-Latency} serverless inference for large language models,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.
- [187] Dejan, “Arduino and mpu6050 accelerometer and gyroscope tutorial,” 2024. [Online]. Available: <https://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/>

- [188] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, “Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence,” *arXiv preprint arXiv:2406.11931*, 2024.
- [189] B. Ding, C. Qin, R. Zhao, T. Luo, X. Li, G. Chen, W. Xia, J. Hu, L. A. Tuan, and S. Joty, “Data augmentation using llms: Data perspectives, learning paradigms and challenges,” in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 1679–1705.
- [190] Z. Wang, G. Xu, and M. Ren, “Llm-generated natural language meets scaling laws: New explorations and data augmentation methods,” *arXiv preprint arXiv:2407.00322*, 2024.
- [191] L. Chen, Y. Tsang, Q. Jing, and L. Sun, “A llm-augmented morphological analysis approach for conceptual design,” 2024.
- [192] Z. Li, L. Si, C. Guo, Y. Yang, and Q. Cao, “Data augmentation for text-based person retrieval using large language models,” *arXiv preprint arXiv:2405.11971*, 2024.
- [193] Z. Wan, H. Shen, X. Wang, C. Liu, Z. Mai, and M. Zhang, “Meda: Dynamic kv cache allocation for efficient multimodal long-context inference,” in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 2485–2497.
- [194] C. Gong, B. Liang, W. Gao, and C. Xu, “Data can speak for itself: Quality-guided utilization of wireless synthetic data,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 209–222.
- [195] X. Cheng, J. Zhang, N. Ding, N. Li, Y. Li, T. Wu, W. Xu, J. Zhang, and Q. Sun, “Integrated ai and communications: A two-way catalysis toward 6g and beyond,” *Journal of Communications and Information Networks*, vol. 10, no. 3, pp. 191–200, 2025.
- [196] C. Y. Lee, P. Yi, M. Fite, T. Rao, S. Achour, and Z. Kapetanovic, “Hypercam: Low-power onboard computer vision for iot cameras,” in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 529–542.
- [197] C. Li, M. Xu, Y. Du, L. Liu, C. Shi, Y. Wang, H. Liu, and Y. Chen, “Practical adversarial attack on wifi sensing through unnoticeable communication packet

- perturbation,” in *Proceedings of the 30th annual international conference on mobile computing and networking*, 2024, pp. 373–387.
- [198] D. Duan, S. Lyu, M. Yuan, H. Xue, T. Li, W. Xu, K. Wu, and G. Xing, “Argus: Multi-view egocentric human mesh reconstruction based on stripped-down wearable mmwave add-on,” in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 1–14.
- [199] A. Orzikulova, D. A. Vasile, C. I. Tang, F. Kawsar, S.-J. Lee, and C. Min, “Bioq: Towards context-aware multi-device collaboration with bio-cues,” in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 504–517.
- [200] Q. Shao, J. Chen, H. M. C. Leung, M. Zhao, R. Xu, J. Liu, L. M. D. García, X. Jiang, M.-P. St-Onge, and X. Zhou, “My pillow knows my sleep: Sleep monitoring with computational fabrics in the pillowcase,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 9, no. 3, pp. 1–26, 2025.
- [201] Q. Shao, J. Liu, E. Bejerano, H. M. Colman, J. Nie, X. Jiang, and X. Zhou, “Joey: Supporting kangaroo mother care with computational fabrics,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 237–251.
- [202] P. Hu, Y. Qian, T. Zheng, A. Li, Z. Chen, Y. Gao, X. Cheng, and J. Luo, “t-readi: Transformer-powered robust and efficient multimodal inference for autonomous driving,” *IEEE Transactions on Mobile Computing*, vol. 24, no. 1, pp. 135–149, 2024.
- [203] T. Do, B. X. Nguyen, Q. D. Tran, E. Tjiputra, T.-C. Chiu, and A. Nguyen, “Lightweight temporal transformer decomposition for federated autonomous driving,” in *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2025, pp. 12 472–12 479.
- [204] F. Ahmad, C. Shin, W. Pang, J. Cashman, B. Leong, and R. Govindan, “idriv-ing: toward safe and efficient infrastructure-directed autonomous driving,” *arXiv preprint arXiv:2207.08930*, 2022.
- [205] J. Cui, C. Tang, J. Holtz, J. Nguyen, A. G. Allievi, H. Qiu, and P. Stone, “Towards natural language communication for cooperative autonomous driving via self-play,” *arXiv preprint arXiv:2505.18334*, 2025.

- [206] M. Zhang, F. Zhang, N. D. Lane, Y. Shu, X. Zeng, B. Fang, S. Yan, and H. Xu, “Deep learning in the era of edge computing: Challenges and opportunities,” *Fog Computing: Theory and Practice*, pp. 67–78, 2020.
- [207] L. Shen, Q. Yang, X. Huang, Z. Ma, Y. Zheng, and C. Xiaoxuan Lu, “One workflow doesn’t fit all: Adaptive workflows for edge ai development,” *Get-Mobile: Mobile Computing and Communications*, vol. 30, no. 1, pp. 11–15, 2026.
- [208] X. Zhang, Z. Huang, E. O. Taga, C. Joe-Wong, S. Oymak, and J. Chen, “Efficient contextual llm cascades through budget-constrained policy learning,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 91 691–91 722, 2024.
- [209] “Reinforcement fine-tuning,” 2024. [Online]. Available: <https://platform.openai.com/docs/guides/reinforcement-fine-tuning>
- [210] H. Wen, S. Tian, B. Pavlov, W. Du, Y. Li, G. Chang, S. Zhao, J. Liu, Y. Liu, Y.-Q. Zhang *et al.*, “Autodroid-v2: Boosting slm-based gui agents via code generation,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 223–235.
- [211] X. Huang, Q. Yang, L. Shen, Z. Ma, and Y. Zheng, “Jailbreaking embodied llms via action-level manipulation,” *arXiv preprint arXiv:2603.01414*, 2026.
- [212] Q. Nguyen, T. Le, H. Nguyen, T. Vo, T. D. Ta, B. Huang, M. N. Vu, and A. Nguyen, “Graspmas: Zero-shot language-driven grasp detection with multi-agent system,” in *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2025, pp. 14 939–14 946.
- [213] Z. Wang, Y. Wang, Z. Wu, H. Ma, Z. Li, H. Qiu, and J. Li, “Cmp: Cooperative motion prediction with multi-agent communication,” *IEEE Robotics and Automation Letters*, 2025.
- [214] S. Zhang, H. Jia, S. Li, T. Dang, Y. Hu, X. Yi, and H. Li, “Position: Human-robot interaction in embodied intelligence demands a shift from static privacy controls to dynamic learning,” *arXiv preprint arXiv:2509.19041*, 2025.
- [215] S. Lee, M. Kang, J. Lee, S. Lee, D. Kim, J. Hong, J. Shin, P. Zhang, and J. Ko, “Llm-based question-answer framework for sensor-driven hvac system interaction,” *arXiv preprint arXiv:2507.04748*, 2025.

- [216] Q. Dong, X. Chen, and M. Satyanarayanan, “Creating edge ai from cloud-based llms,” in *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*, 2024, pp. 8–13.
- [217] R. Choi, T. Kim, S. Park, S. Yoo, J. G. Kim, and S.-J. Lee, “Collective voice: Recovered-peer support mediated by an llm-based chatbot for eating disorder recovery,” *arXiv preprint arXiv:2509.15289*, 2025.
- [218] R. Choi, T. Kim, S. Park, J. G. Kim, and S.-J. Lee, “Private yet social: How llm chatbots support and challenge eating disorder recovery,” in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–19.
- [219] Z. Zhang, A. Metzger, Y. Mei, F. Hähnlein, Z. Englhardt, T. Cheng, G. D. Abowd, S. Patel, A. Schulz, and V. Iyer, “Towards autonomous sustainability assessment via multimodal ai agents,” *arXiv preprint arXiv:2507.17012*, 2025.
- [220] Z. Ma, L. Shen, X. Huang, and Y. Zheng, “Poster: Llmware: An llm-powered robust and efficient android malware detection framework,” in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 4737–4739.
- [221] L. Shen and Y. Zheng, “Federated self-evolving embodied ai agents,” in *IEEE INFOCOM 2026-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2026, pp. 1–6.
- [222] T. Zheng, A. Li, Z. Chen, H. Wang, and J. Luo, “Autofed: Heterogeneity-aware federated multimodal learning for robust autonomous driving,” in *Proceedings of the 29th annual international conference on mobile computing and networking*, 2023, pp. 1–15.
- [223] H. Pan, X. Zhao, Y. Jiang, L. He, B. Wang, and Y. Shu, “Fedvlp: Visual-aware latent prompt generation for multimodal federated learning,” *Computer Vision and Image Understanding*, vol. 259, p. 104442, 2025.
- [224] M. Prabu, S. Sountharajan, E. Suganya, and D. P. Bavirisetti, “Contribution of python to improving efficiency in artificial intelligence and advancing automation capabilities,” in *Smart Computing Techniques in Industrial IoT*. Springer, 2024, pp. 201–218.
- [225] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, “Judging llm-as-a-judge with mt-bench and chatbot

- arena,” *Advances in neural information processing systems*, vol. 36, pp. 46 595–46 623, 2023.
- [226] S. Han, Q. Zhang, W. Jin, and Z. Xu, “Llm multi-agent systems: Challenges and open problems,” *arXiv preprint arXiv:2402.03578*, 2024.
- [227] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, “Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices,” in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 81–93.
- [228] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, and W.-m. Hwu, “Xsp: Across-stack profiling and analysis of machine learning models on gpus,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 326–327.
- [229] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, “Self-consistency improves chain of thought reasoning in language models,” *arXiv preprint arXiv:2203.11171*, 2022.
- [230] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, H. Wang *et al.*, “Retrieval-augmented generation for large language models: A survey,” *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, p. 32, 2023.
- [231] Z. Zhang, P. Dash, Y. C. Hu, Q. Xu, J. Li, and H. Guan, “Dissecting the impact of mobile dvfs governors on llm inference performance and energy efficiency,” *arXiv preprint arXiv:2507.02135*, 2025.
- [232] Y. Liu, Q. Xu, and Y. C. Hu, “Cronus: Efficient llm inference on heterogeneous gpu clusters via partially disaggregated prefill,” *arXiv preprint arXiv:2509.17357*, 2025.
- [233] Z. Wan, Z. Dou, C. Liu, Y. Zhang, D. Cui, Q. Zhao, H. Shen, J. Xiong, Y. Xin, Y. Jiang *et al.*, “Srpo: Enhancing multimodal llm reasoning via reflection-aware reinforcement learning,” *arXiv preprint arXiv:2506.01713*, 2025.
- [234] R. Yuan, R. M. Gower, and A. Lazaric, “A general sample complexity analysis of vanilla policy gradient,” in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 3332–3380.
- [235] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *arXiv preprint arXiv:1506.02438*, 2015.

- [236] V. Konda and J. Tsitsiklis, “Actor-critic algorithms,” *Advances in neural information processing systems*, vol. 12, 1999.
- [237] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology*, vol. 108, pp. 115–138, 2019.
- [238] “o4-mini,” 2024. [Online]. Available: <https://platform.openai.com/docs/models/o4-mini>
- [239] chroma-core contributors, “chroma: Data infrastructure for ai,” <https://github.com/chroma-core/chroma>, 2026, gitHub repository, accessed 2026-04-19.
- [240] Pylint contributors, “Pylint.” [Online]. Available: <https://pylint.readthedocs.io/en/latest>
- [241] X. Yue, Y. Ni, K. Zhang, T. Zheng, R. Liu, G. Zhang, S. Stevens, D. Jiang, W. Ren, Y. Sun *et al.*, “Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2024, pp. 9556–9567.
- [242] T. Matuszka, I. Barton, Á. Butykai, P. Hajas, D. Kiss, D. Kovács, S. Kunsági-Máté, P. Lengyel, G. Németh, L. Pető *et al.*, “aimotive dataset: A multi-modal dataset for robust autonomous driving with long-range perception,” *arXiv preprint arXiv:2211.09445*, 2022.
- [243] ONNX Contributors, “onnx/onnx: Open standard for machine learning interoperability,” <https://github.com/onnx/onnx>, 2025, gitHub repository, accessed 2026-04-19.
- [244] A. Sarkar and I. Drosos, “Vibe coding: programming through conversation with artificial intelligence,” *arXiv preprint arXiv:2506.23253*, 2025.
- [245] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language agents with verbal reinforcement learning,” *Advances in neural information processing systems*, vol. 36, pp. 8634–8652, 2023.
- [246] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, “Tree of thoughts: Deliberate problem solving with large language models,” *Advances in neural information processing systems*, vol. 36, pp. 11 809–11 822, 2023.

- [247] X. Ouyang, Z. Xie, J. Zhou, J. Huang, and G. Xing, “Clusterfl: a similarity-aware federated learning system for human activity recognition,” in *Proceedings of the 19th annual international conference on mobile systems, applications, and services*, 2021, pp. 54–66.
- [248] X. Kupers, J. K. Brinke, R. Bemthuis, and O. D. Incel, “Towards edge-based idle state detection in construction machinery using surveillance cameras,” in *Intelligent Systems Conference*. Springer, 2025, pp. 402–418.
- [249] D. Romero, Y. Chandio, F. Anwar, and S. Elmalaki, “Murmur: A multimodal sensing framework for automated group behavior analysis in mixed reality,” *arXiv preprint arXiv:2507.11797*, 2025.
- [250] A. Ghosh, N. Garg, and N. Roy, “Deepsync: A learning framework for pervasive localization using code synchronization on compressed cellular spectrum,” *arXiv preprint arXiv:2505.05783*, 2025.
- [251] M. Kalanadhabhatta, A. M. Santana, D. Ganesan, T. Rahman, and A. Grabell, “Extracting multimodal embeddings via supervised contrastive learning for psychological screening,” in *2022 10th International Conference on Affective Computing and Intelligent Interaction (ACII)*. IEEE, 2022, pp. 1–8.
- [252] “Hugging face,” 2024. [Online]. Available: <https://huggingface.co/>
- [253] “Github,” 2024. [Online]. Available: <https://github.com/>
- [254] J. Singh, R. Magazine, Y. Pandya, and A. Nambi, “Agentic reasoning and tool integration for llms via reinforcement learning,” *arXiv preprint arXiv:2505.01441*, 2025.
- [255] ONNX Runtime developers, “Onnx runtime,” <https://github.com/microsoft/onnxruntime>, Nov. 2018. [Online]. Available: <https://onnxruntime.ai>
- [256] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [257] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, “{TVM}: An automated {End-to-End} optimizing compiler for deep learning,” in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 578–594.

- [258] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, K. Kawachiya *et al.*, "Compiling onnx neural network models using mlir," *arXiv preprint arXiv:2008.08272*, 2020.
- [259] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.
- [260] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, "Deformable detr: Deformable transformers for end-to-end object detection," *arXiv preprint arXiv:2010.04159*, 2020.
- [261] S.-g. Lee, W. Ping, B. Ginsburg, B. Catanzaro, and S. Yoon, "Bigvgan: A universal neural vocoder with large-scale training," *arXiv preprint arXiv:2206.04658*, 2022.
- [262] 2024. [Online]. Available: [https://onnx.ai/onnx/operators/onnx\\_\\_Resize.html](https://onnx.ai/onnx/operators/onnx__Resize.html)
- [263] X. Huang, L. Shen, Z. Ma, and Y. Zheng, "Poster: Towards privacy-preserving and personalized smart homes via tailored small language models," in *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, 2025, pp. 1332–1334.
- [264] X. Li, S. Yu, L. Shen, J. Zhang, Y. Zheng, and Y. Xie, "Autoran: Llm-driven natural language programming for agile xapp development," *arXiv preprint arXiv:2603.18604*, 2026.
- [265] W. Sun, C. Fang, Y. Miao, Y. You, M. Yuan, Y. Chen, Q. Zhang, A. Guo, X. Chen, Y. Liu *et al.*, "Abstract syntax tree for programming language understanding and representation: How far are we?" *arXiv preprint arXiv:2312.00413*, 2023.
- [266] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2023.
- [267] 2024. [Online]. Available: <https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html#basic-graph-optimizations>
- [268] P. Merz and B. Freisleben, "Greedy and local search heuristics for unconstrained binary quadratic programming," *Journal of heuristics*, vol. 8, no. 2, pp. 197–213, 2002.

- [269] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, “Gpt-4o system card,” *arXiv preprint arXiv:2410.21276*, 2024.
- [270] “Openai text-embedding-ada-002,” <https://platform.openai.com/docs/guides/embeddings/embedding-models>, 2022.
- [271] K. N. Khan, A. Khalid, Y. Turkar, K. Dantu, and F. Ahmad, “Vrf: Vehicle road-side point cloud fusion,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 547–560.
- [272] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [273] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [274] 2024. [Online]. Available: <https://huggingface.co/firdhokk/speech-emotion-recognition-with-openai-whisper-large-v3>
- [275] E. J. Lok, “Toronto emotional speech set (tess),” <https://www.kaggle.com/datasets/ejlok1/toronto-emotional-speech-set-tess>, 2018, kaggle dataset, accessed 2026-04-19.
- [276] Z. Zhang, C. Hu, S. Lye, and C. Lv, “A vlm-drone system for indoor navigation assistance with semantic reasoning for the visually impaired,” in *2025 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2025, pp. 1260–1265.
- [277] H. Lu, W. Liu, B. Zhang, B. Wang, K. Dong, B. Liu, J. Sun, T. Ren, Z. Li, H. Yang *et al.*, “Deepseek-vl: towards real-world vision-language understanding,” *arXiv preprint arXiv:2403.05525*, 2024.
- [278] Keras Team, “keras,” 2026, gitHub repository, accessed 2026-04-19. [Online]. Available: <https://github.com/keras-team/keras>
- [279] 2024. [Online]. Available: [https://huggingface.co/OpenDILabCommunity/webpage\\_element\\_detection](https://huggingface.co/OpenDILabCommunity/webpage_element_detection)
- [280] 2024. [Online]. Available: [https://onnx.ai/onnx/operators/onnx\\_\\_Upsample.html](https://onnx.ai/onnx/operators/onnx__Upsample.html)

- [281] D. Cai, S. Wang, C. Peng, Z. Zhang, Z. Lu, T. Qi, N. D. Lane, and M. Xu, “Ubiquitous memory augmentation via mobile multimodal embedding system,” *Nature Communications*, vol. 16, no. 1, p. 5339, 2025.
- [282] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *The eleventh international conference on learning representations*, 2022.
- [283] S. Lee and S. Nirjon, “Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 15–29.
- [284] R. Liu, Y. Leng, S. Tian, S. Hu, C.-F. Chen, and S. Yao, “Dynaspa: Exploiting spatial sparsity for efficient dynamic dnn inference on devices,” in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, 2024, pp. 422–435.
- [285] H. Jia, Y. D. Kwon, A. Orsino, T. Dang, D. Talia, and C. Mascolo, “Tinytta: Efficient test-time adaptation via early-exit ensembles on edge devices,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 43 274–43 299, 2024.
- [286] J. Park and J. Ko, “Fedhm: Practical federated learning for heterogeneous model deployments,” *ICT Express*, vol. 10, no. 2, pp. 387–392, 2024.
- [287] N. Said and O. Landsiedel, “Edgeboost: Confidence boosting for resource constrained inference via selective offloading,” *Computer Networks*, vol. 269, p. 111437, 2025.
- [288] P. Rathje and O. Landsiedel, “Aladin: Autonomous linear antenna delay inference on resource-constrained ultra-wideband devices,” in *Proceedings of the 2023 INTERNATIONAL CONFERENCE ON EMBEDDED WIRELESS SYSTEMS AND NETWORKS*, 2023, pp. 181–191.
- [289] Y. Sun and J. Ortiz, “An ai-based system utilizing iot-enabled ambient sensors and llms for complex activity tracking,” *arXiv preprint arXiv:2407.02606*, 2024.
- [290] X. Huang, L. Shen, Z. Ma, and Y. Zheng, “Towards privacy-preserving and personalized smart homes via tailored small language models,” *IEEE Transactions on Mobile Computing*, 2026.
- [291] X. Zhang, Z. Huang, C. Ni, Z. Xiong, J. Chen, and S. Oymak, “Making small language models efficient reasoners: Intervention, supervision, reinforcement,” *arXiv preprint arXiv:2505.07961*, 2025.

- [292] L. Jiao, C. Gao, Y. Yang, C. Zhou, Y. Huang, X. Chen, and Y. Li, “Analyzing and modeling llm response lengths with extreme value theory: Anchoring effects and hybrid distributions,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 32 980–32 990.
- [293] S. Biswas, M. N. H. Khan, and B. Islam, “Owl: Geometry-aware spatial reasoning for audio large language models,” *arXiv preprint arXiv:2509.26140*, 2025.
- [294] G. Burr, H. Tsai, W. Simon, I. Boybat, S. Ambrogio, C.-E. Ho, Z.-W. Liou, M. Rasch, J. Büchel, P. Narayanan *et al.*, “Design of analog-ai hardware accelerators for transformer-based language models,” in *2023 International Electron Devices Meeting (IEDM)*. IEEE, 2023, pp. 1–4.
- [295] X. Wang, T. Dang, X. Zhang, V. Kostakos, M. J. Witbrock, and H. Jia, “Healthslm-bench: Benchmarking small language models for mobile and wearable healthcare monitoring,” *arXiv preprint arXiv:2509.07260*, 2025.
- [296] H.-s. Kim, J. Hong, K. Jang, H. Lee, H. K. Ahn, and H.-W. Shin, “Per-second, explainable obstructive sleep apnea detection from multimodal time-series using vision transformer,” 2025.
- [297] X. Wang, S. Alam, Z. Wan, H. Shen, and M. Zhang, “Svd-llm v2: Optimizing singular value truncation for large language model compression,” in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 4287–4296.
- [298] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang *et al.*, “Repoagent: An llm-powered open-source framework for repository-level code documentation generation,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2024, pp. 436–464.
- [299] Z. Xu, H. Xu, Z. Lu, Y. Zhao, R. Zhu, Y. Wang, M. Dong, Y. Chang, Q. Lv, R. P. Dick *et al.*, “Can large language models be good companions? an llm-based eyewear system with conversational common ground,” *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 8, no. 2, pp. 1–41, 2024.
- [300] Y. Ishibashi and Y. Nishimura, “Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization,” *arXiv preprint arXiv:2404.02183*, 2024.

- [301] L. Shen and Y. Zheng, “Poster: Towards federated embodied ai with feai,” in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 599–600.
- [302] Z. Li, W. Hua, H. Wang, H. Zhu, and Y. Zhang, “Formal-llm: Integrating formal language and natural language for controllable llm-based agents,” *arXiv preprint arXiv:2402.00798*, 2024.