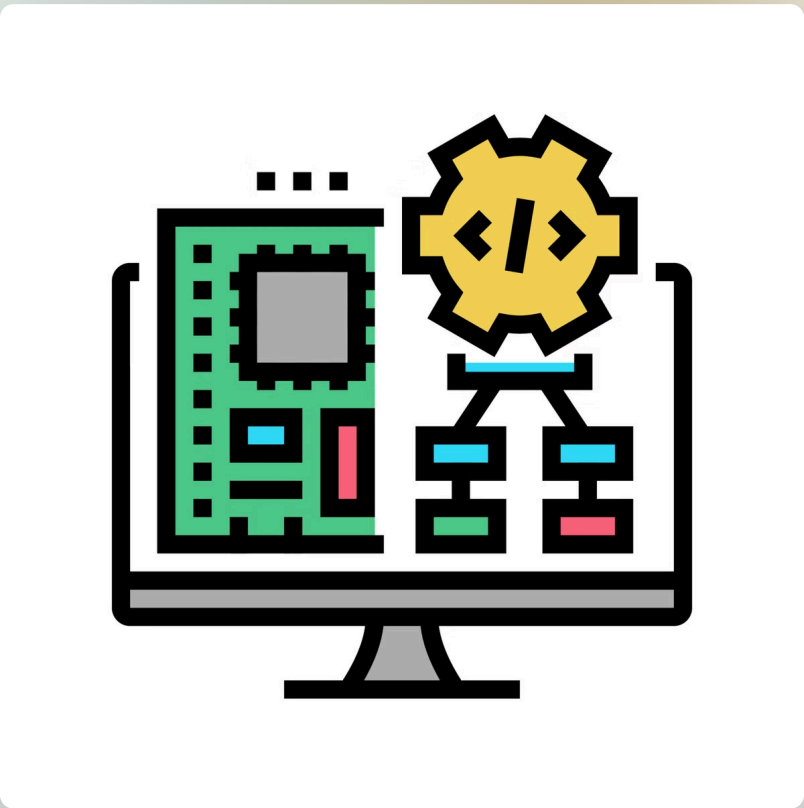


Dynamic Queries

Adaptive QA across Edge and Cloud

A intelligent system to switch between models running on cloud to models running on edge during runtime in order to produce accurate and environmentally sustainable results for the end-user.

ESW Project by Team Bond Brothers:
Krishak Aneja | Saiyam Jain | Varun Gupta | Gracy Garg

Project Overview: Goals and Objectives

1

Exploration of the QIDK

To test and use the hardware provided to us and run various models on it whilst trying to understand the capabilities of the kit. Further, learn to apply it to more advanced use cases.

2

Dynamic Selection of Models

To design an system capable of intelligently selecting a model to execute the query whilst optimizing power consumption,speed and accuracy.

Environmental impact (energy wastage) has been considered as a priority metric in the selection of the most optimized model.

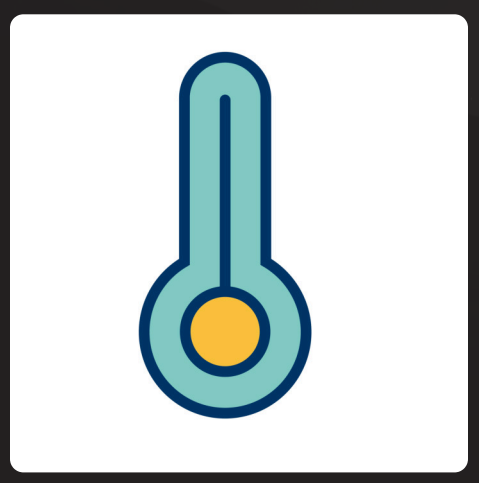
3

User Interface and Analysis

To develop an aesthetically pleasing and intuitive user interface that enhances user interaction with the QIDK platform. The interface will feature clean layouts, and responsive controls for seamless model execution and performance analysis.

Factors Considered

1. CPU Usage
2. Battery Level
3. Temperature
4. User Feedback
5. Token Count



Switching Logic

The switching logic takes inspiration from the MLFQ (Multi-Level Feedback Queue) scheduling policy to determine the most optimal model for running a query based on previous performance as well as a Scoring metric that decides if the Model is to be demoted or promoted.

The 'Worth' score.

Each model is assigned a tokenFactor and an eFactor that represent the model's affinity for the token size of the content and the environmental sustainability level of the model respectively. A more powerful model can handle more tokens but also takes up more energy and is thereby less environmentally sustainable.

After each successful execution, a dynamic scoring metric computes a score for the model based on key metrics (Battery, CPU, User feedback, tokenFactor, and eFactor). The weight given to each metric adapts based on the system's current state, ensuring context-sensitive decision-making.

The score is then compared to the Estimated Moving Average (EMA) score and the model is reassigned to a queue accordingly. EMA helps smooth the fluctuations in scores over time, avoiding sudden priority changes.

A robust switching logic that uses reinforcement learning ensures that the system is adaptive, scalable, and capable of handling varying conditions in real time.

```
tModel() {  1 usage
yQueue.isEmpty()) return highPriorityQueue.peek();
Queue.isEmpty()) return midPriorityQueue.peek();
ityQueue.peek();


ulateScore(int batteryLevel, float batteryConsumption, float temperature, float cpuUsage, float tokenFactor, double feedback, flo
ttery = (batteryLevel < 30) ? 0.4 : 0.2;
uUsage = (temperature > 40) ? 0.4 : 0.2;
edback = Math.round((1.0 - weightBattery - weightCpuUsage)* 10) / 10.0;

ores:
core = 1.0 - (batteryConsumption / 100.0);
 = 1.0 - (cpuUsage / 100.0);

(batteryScore * weightBattery) + (cpuScore * weightCpuUsage) + (feedback * weightFeedback);
ulating...", msg: "eFactor:" + eFactor + " tokenFactor:" +tokenFactor);
ery", msg: " batteryScore:"+batteryScore+ " weightBattery:" + weightBattery);
, msg: " cpuScore:"+cpuScore+ " weightCpu:" + weightCpuUsage );
back", msg: " feedback:"+feedback+ " weightFeedback" + weightFeedback);
 tokenFactor * score;


MA(double newScore) { emaScore = ALPHA * newScore + (1 - ALPHA) * emaScore; }
```

Score Calculator

Switching Logic

The MLFQ.

```
94  public class QaClient {  4 usages
116     private double emaScore = 0.0;  7 usages
117     private static final double ALPHA = 0.25;  2 usages
118
119     private static final int PRIORITY_HIGH = 0;  3 usages
120     private static final int PRIORITY_MID = 1;  3 usages
121     private static final int PRIORITY_LOW = 2;  2 usages
122
123     private Queue<Model> highPriorityQueue = new LinkedList<>();  7 usages
124     private Queue<Model> midPriorityQueue = new LinkedList<>();  9 usages
125     private Queue<Model> lowPriorityQueue = new LinkedList<>();  6 usages
126
127     private static final int LocalNo = 0;  3 usages
128     private static final int CloudNo = 1;  4 usages
129
130     private static Model prevModel;  18 usages
131     private static float prevCpuUsage = 0.0f;  5 usages
132     private static float prevBatteryConsumption = 0.0f;  2 usages
133
134     private static int agingFactor = 3;  1 usage
135     private static int runsSinceLastBoost = 0;  3 usages
136     private static int boostInterval = 100;  1 usage
137     private static int tokenThreshold = 300;  1 usage
138
139     private class Model {...}
163
164     public QaClient(Context context) {...}
171
172 @   private void promoteModel(Model model) {  1 usage
173         if(model.priority > PRIORITY_HIGH) {
174             model.priority--;
175             moveModelToQueue(model);
176         }
177     }
178
179 @   private void demoteModel(Model model) {  2 usages
180         if(model.priority < PRIORITY_LOW) {
181             model.priority++;
182             moveModelToQueue(model);
183         }
184     }
```

The system uses an MLFQ-inspired strategy to dynamically select models based on their performance and resource efficiency.

- **Selection**: Models are chosen from high, medium, or low-priority queues, with higher-priority models given preference.
- **Demotion**: If a model under performs or exceeds thresholds (e.g., low EMA scores or repeated rejections), it is demoted to a lower-priority queue.
- **Aging**: Frequently used models move within queues to prevent dominance and ensure fair usage.
- **Priority Boost**: At regular intervals, all models are reassigned to the high-priority queue to avoid starvation.
- **Rejection Handling**: Models rejected more than three times are demoted, resetting execution statistics to avoid inefficient retries.

This robust strategy balances performance and efficiency, adapting dynamically to changing conditions.

Models Used

We have used 4 different QnA models for now , 2 of these are local and 2 are cloud based models.

QIDK: QnA Model

Runs Locally. This is the Question Answering model provided by Qualcomm.

Gemini

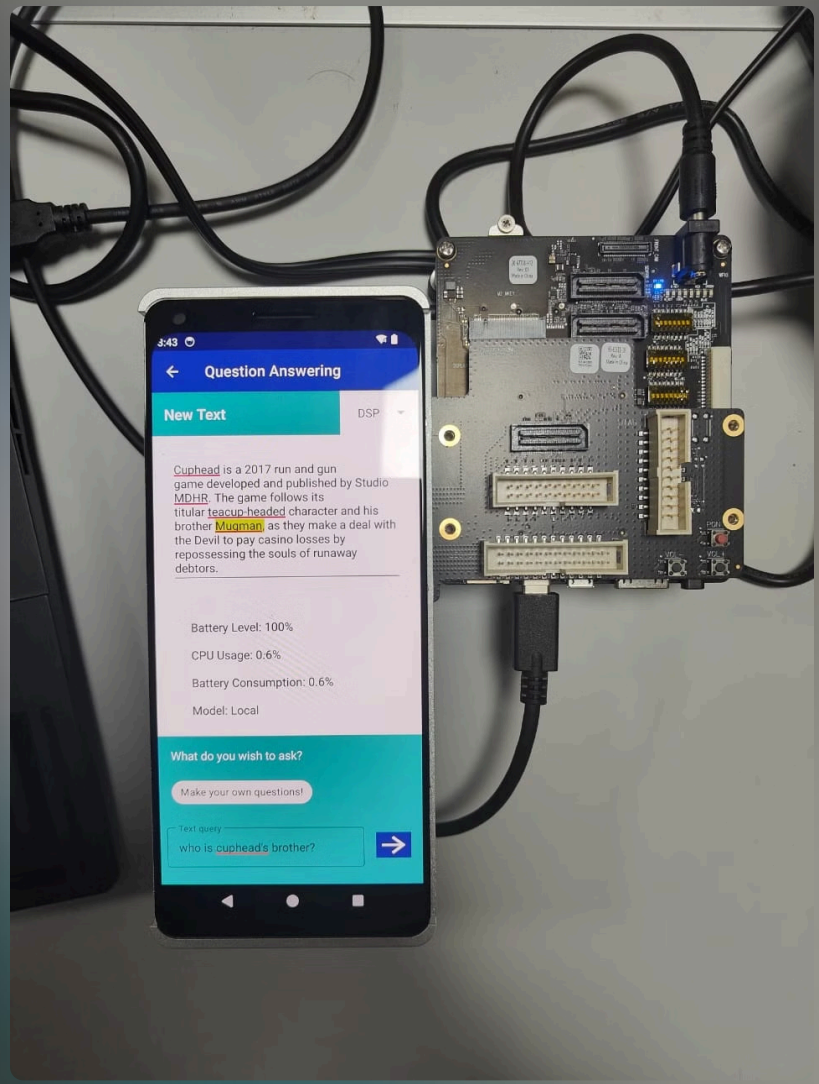
Cloud model. Answered are fetched using API calls.

Bert

Also runs locally.

Ollama

Cloud Model. Answers fetched from the model running on a PC system as server.



Details of the QIDK model:

Question Answering (QA) is one of the common and challenging Natural Language Processing tasks.

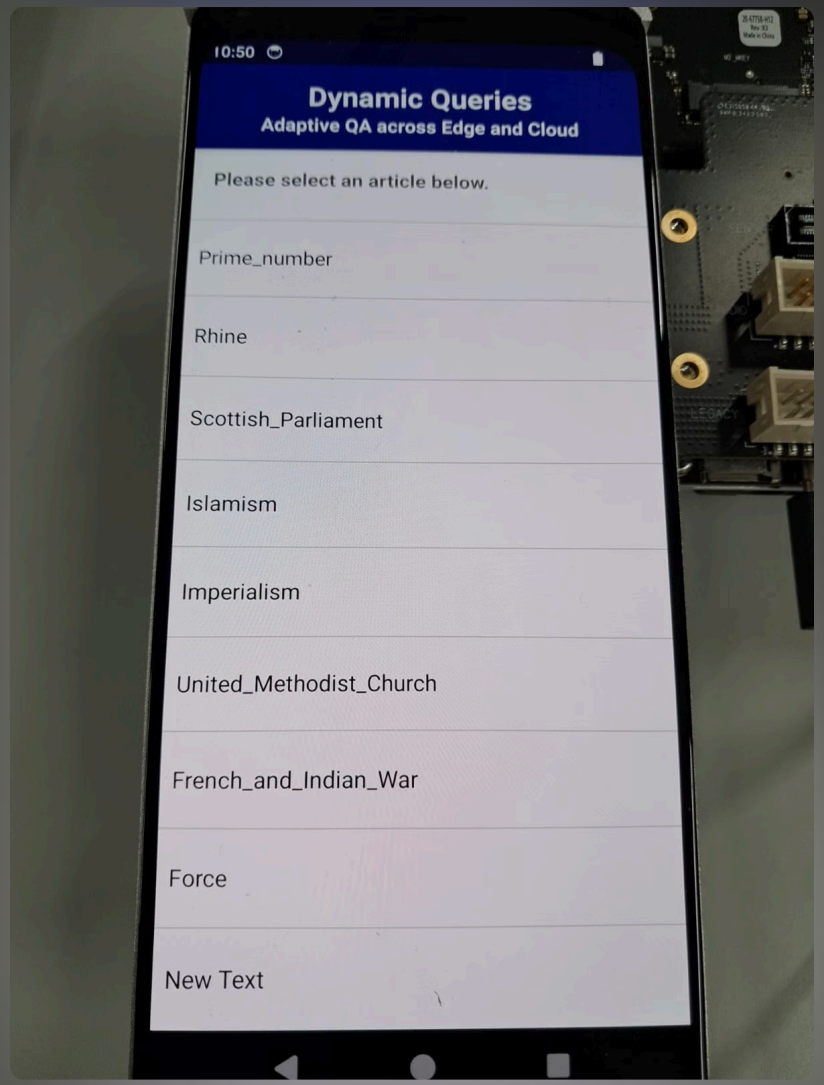
- We have used the sample Android application for On-Device Question Answering based on ICLR 2020 Electra Transformer model accelerated using Qualcomm Neural Processing SDK for AI framework, as a base framework of our project.
- Model used in this project is : https://huggingface.co/mrm8488/electra-small-finetuned-squadv2
- https://huggingface.co/mrm8488/electra-small-finetuned-squadv2 is a small, efficient and mobile friendly Transformer model fine-tuned on SQUAD v2.0 dataset for **Q&A** downstream task.

Optimization using caching

The model-switching logic has been further optimized by implementing a caching mechanism. Models responses are loaded and stored in a cache. Subsequent requests tries to retrieve it from the cache,and if found it significantly reduc the overhead of reloading models and improving efficiency when switching between different machine learning models.

Further, the cache also stores frequently used inputs and their corresponding results, allowing the system to quickly handle repeated or similar queries without reprocessing them. As a result, processing time is reduced, and response times are significantly faster. This optimization ensures the system operates more efficiently, providing quicker and more responsive interactions for users.

Cache Memory



Choosing Content

The user may choose to ask questions based on any of the given sample texts or alternatively, add their own content in the 'New Text section' and query it.

Sample questions are also provided with the sample text to facilitate testing.

Modes of Operation:

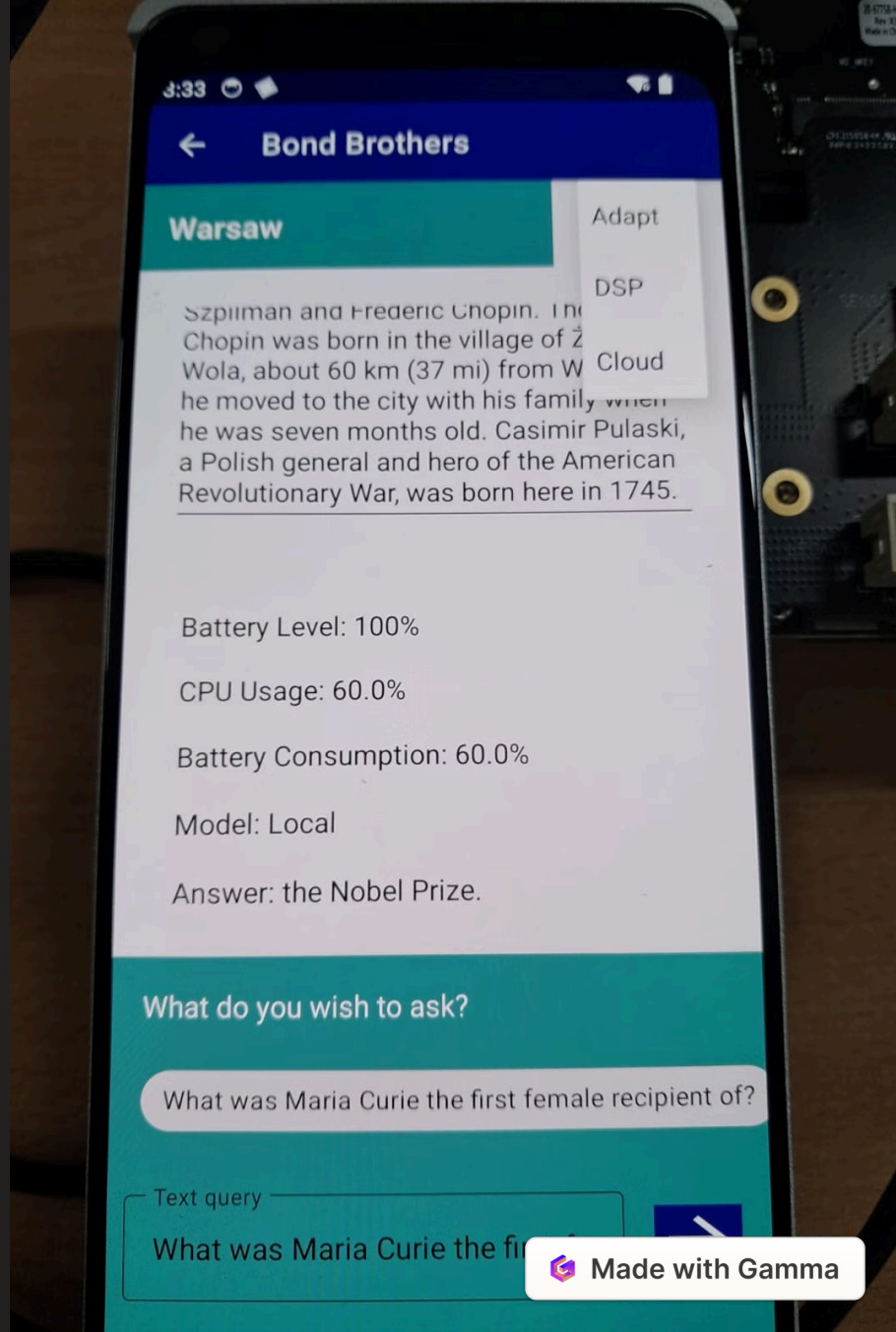
The application provides two modes of operation:

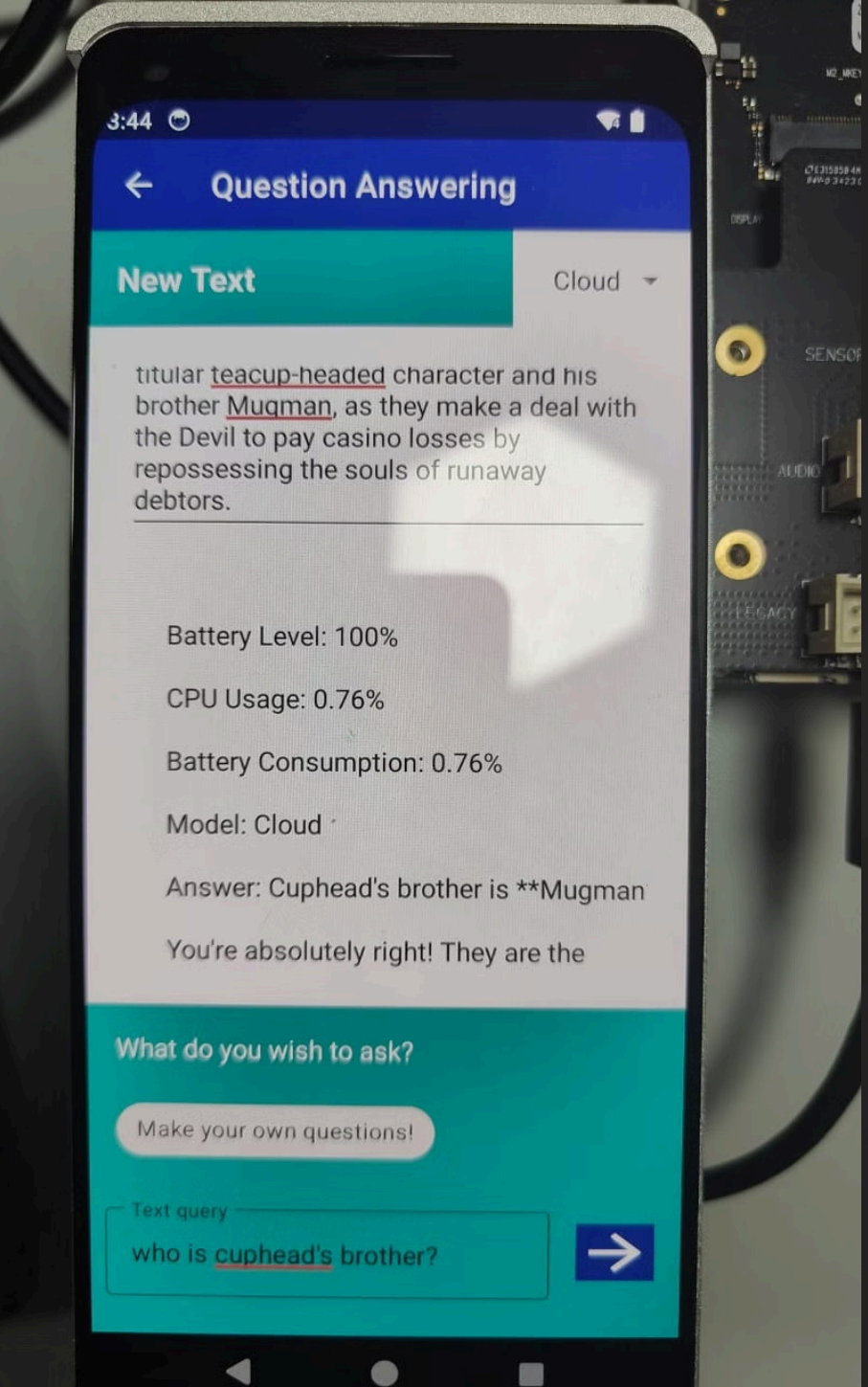
*** Adapt**

The default mode. Uses the implemented switching logic to dynamically choose which model should run.

*** Manual**

Choose to run an individual model only. The EMA score is still updated for each run.





Conclusion

The application effectively integrates dynamic model switching, an intuitive user interface, and caching mechanisms to optimize performance and user experience. By leveraging real-time metrics such as input size, power usage, and user feedback, it intelligently selects the most suitable model for each query, ensuring efficient execution across diverse use cases.

The app can fetch answers to queries effectively, handling both small and large prompts by integrating the benefits of simple and larger models, switching seamlessly between edge and cloud resources. The user interface enhances accessibility with clean layouts and responsive controls, while the caching system reduces overhead by storing frequently queried inputs.

Additionally, connecting with a PC system allows the collection of key metrics, providing insights for continuous improvement and adaptability and allows graphing of data. This comprehensive approach ensures scalability, efficiency, and environmental sustainability.