An Exchange-based AIoT Platform for Fast AI Application Development

Yu-Cheng Liang
Department of Computer Science,
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
ycliang.c@nycu.edu.tw

Kun-Ru Wu
Department of Computer Science,
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
wufish@nycu.edu.tw

Kit-Lun Tong
School of Computing Sciences,
University of East Anglia
Norwich, UK
k.tong@uea.ac.uk

Yi Ren
School of Computing Sciences,
University of East Anglia
Norwich, UK
e.ren@uea.ac.uk

Yu-Chee Tseng
Department of Computer Science,
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
Miin Wu School of Computing,
National Cheng Kung University
Tainan, Taiwan
yctseng@cs.nycu.edu.tw

ABSTRACT

AIoT is the combination of Internet of Things (IoT) and Artificial Intelligence (AI) technologies. While IoT emphasizes more on scalable and efficient communications, AI focuses more on reproducing human capabilities such as recognition and forecasting. An efficient AIoT platform may not be obtained directly from integrating existing IoT and AI serving platforms by considering the AIoT service reproduction and evolution. In this work, we propose an AIoT platform that empowers developers to build sophisticated and scalable applications. Our platform is derived based on exchange-based RabbitMQ broker and Advanced Message Queuing Protocol (AMQP) to facilitate the communications among heterogeneous data sources and AI models. By incorporating an AMQP broker, it supports diverse data exchanges, AI models chaining, and flexible message routing and processing. AI models can be deployed efficiently through containerization with flexible and shared data paths to facilitate computations. Hence, developers can focus on service and application requirements. We also present a case study in smart healthcare to validate our design.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; *Software system structures*; • **Networks** → Network services.

KEYWORDS

AIoT; Application Platform; Advanced Message Queuing Protocol (AMQP); AI Models Chaining; Service Configuration; Service-Oriented Architecture

ACM Reference Format:
Yu-Cheng Liang, Kun-Ru Wu, Kit-Lun Tong, Yi Ren, and Yu-Chee Tseng. 2023. An Exchange-based AIoT Platform for Fast AI Application Development. In *Proceedings of the 19th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet '23), October 30 – November 3, 2023, Montreal, Canada.*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3616391.3622770

1 INTRODUCTION

As IoT technology continues to develop, more devices and equipment are being connected to the Internet, creating an extensive data network. However, simple IoT technology often faces challenges in converting collected data into useful information and knowledge. The emergence of artificial intelligence technologies offers new possibilities to solve this problem. AIoT (Artificial Intelligence of the Things) combines IoT with AI technologies, thus creating many intelligent applications [2, 9, 21, 25, 26, 31, 38, 39].

A powerful and flexible AIoT platform is essential for achieving complex and diverse AIoT applications. On the market, several IoT platforms are available, including IBM Node-RED [11], AWS IoT [35], Microsoft Azure IoT [28], and Google Cloud IoT [18], which offer a variety of IoT services, such as device management, data collection and analysis, and support for multiple communication protocols. On the other hand, there also exist various AI model serving platforms, such as TensorFlow Serving [30], PyTorch Serving [12], Clipper [13], and BentoML [4], which enable developers to deploy and execute their AI models at scale. However, directly integrating an existing IoT platform and an existing AI model serving platform may present several challenges. First, an IoT platform may lack knowledge to analyze and process large amounts of data, particularly when integrating with AI models. Second, interoperability issues may arise when integrating different data formats and communication protocols between IoT and AI systems. To illustrate, IoT devices may generate data in JSON and XML formats, while AI models may require input data in a specific format, such as CSV or Tensor. Third, the lack of customization and flexibility in existing platforms may not meet the specific requirements of an AIoT application. Therefore, a dedicated AIoT platform designed to address these challenges is necessary to meet the requirements of AIoT applications [43].

Inspired by the above observations, we propose an AIoT platform that empowers users to build sophisticated and scalable AIoT applications. Our platform features a distributed and modular architecture that provides flexibility and scalability. AI models and applications are deployed through a container-based microservice architecture, which simplifies system management and updates. Our platform allows an AIoT developer to integrate several modular AI models to accomplish complex inference tasks. Moreover, we use RabbitMQ broker [32] and AMQP protocol [40] to facilitate the communications among heterogeneous data sources and AI models, thus greatly relieving the pains of AIoT developers in handling the interoperability issue.

The design of our AIoT platform has several features that conquer the challenges faced by existing platforms. First, it enables users to easily integrate multiple IoT devices and AI models into a single application, reducing the complexity and time in developing an AIoT application. Second, it allows for flexible and efficient scaling of services and straightforward management of AI model version update issue. Third, it facilitates the building of complex inference tasks, simplifying the development process and allowing for the integration of multiple types of AI models. We also present a smart healthcare application example to demonstrate the advantage of these designs.

The rest of this paper is organized as follows. Section 2 presents related works. Section 3 introduces our platform design. Section 4 demonstrates a case study in smart healthcare. Section 5 makes comparisons to other works. Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 Background on Messaging Protocols

Messaging protocols play a critical role in the architecture of any IoT and AIoT platform. It acts as the means of communication between devices and the platform. The work [1, 29, 36] provided a comprehensive comparison and introduction to the messaging protocols that are commonly used in IoT systems. HTTP (HyperText Transfer Protocol) is a web-based request-response protocol widely adopted in the context of World Wide Web (WWW). Due to its reliability and ubiquity, HTTP has also been applied to IoT solutions, especially when a direct Internet connection is possible. MQTT (Message Queuing Telemetry Transport) is a lightweight publish-subscribe messaging protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. In MQTT, a central broker handles the distribution of messages to clients. AMQP (Advanced Message Queuing Protocol) is a protocol designed for robust, flexible, and open messaging. It supports both point-to-point and publish-subscribe models, making it a versatile solution for many applications. Unlike HTTP and MQTT, AMQP provides strong guarantees for message delivery, including confirmations and the ability to resume interrupted transfers. The study [6] used the AMQP protocol to implement low-cost devices in a simulated factory to control industrial processes and integrate shopfloor communications. Reference [34] presented a framework for developing digital twins that combine machine learning, IoT, and 3D visualization, and used AMQP for message exchange between components. Tab. 1 compares these three key messaging protocols in terms of design pattern, data distribution, messaging overhead, message routing, and encoding format. AMQP offers high reliability, flexible message routing capabilities, and a balanced trade-off between feature-richness and performance, making it an excellent choice for AIoT applications.

Table 1: Comparison of HTTP, MQTT, and AMQP protocols.

	HTTP	MQTT	AMQP
Design Pattern	Request-Response	Publish-Subscribe	Request-Response; Publish-Subscribe
Data Distribution	1-to-1	1-to-N; N-to-N	1-to-1; N-to-N
Interoperability	High	Low	Moderate
Messaging Overhead	High	Low	Moderate
Message Routing	Direct	Topic-based	Direct; Topic; Fanout; Headers
Encoding Format	Text	Binary	Binary

2.2 IoT platforms

There are two main IoT architectures widely used to build IoT applications: the Service-oriented Architecture (SOA-IoT) and the Microservice-IoT architecture.

SOA-IoT is an extension of the traditional SOA, which utilizes service-oriented design principles to support the development of IoT applications [14]. Reference [7] proposed a smart IoT communication system manager based on SOA principles as a cost-effective irrigation controller. SOA and IoT have been used in [16] to implement an M2M application in the field of road traffic management. The work [3] reviewed the applications of SOA in home-based patient care within the health industry and examined the potential of IoT in telemedicine. However, SOA-IoT requires a centralized architecture that needs to be fully designed and planned during the development phase, which may not satisfy the dynamic nature of AIoT applications.

Microservice IoT is an approach that utilizes the microservice architecture to enhance scalability and flexibility for IoT applications. This approach splits a monolithic application into a set of distributed services that are highly decoupled, thus reducing maintenance and update efforts and improving modularity [5]. The study [22] explored the use of microservice architecture in building a smart city IoT platform and discussed its benefits and challenges compared to SOA approaches. In [37], the authors proposed an IoT platform based on microservice architecture and validated it on a smart farming application. The study [10] introduced a platform that leverages a microservice IoT-Big Data architecture for the distributed sharing of multidisciplinary simulation models. However, these approaches only focus on IoT applications. For complex AIoT applications composed of a multitude of AI models, inter-service communications often need to integrate IoT and AI services. This could be a challenge when applying the current microservice IoT architecture to AIoT applications.

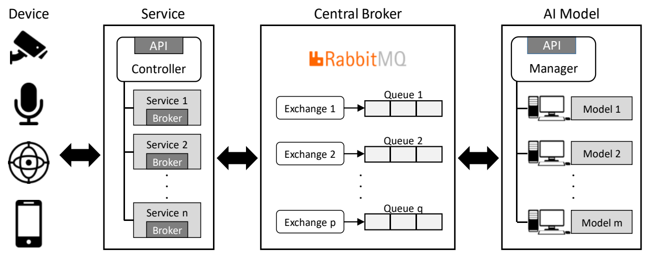

Figure 1: System architecture.

2.3 AI Model Serving Platforms

An AI model serving platform is designed to manage the deployment of machine learning models in a production environment. Such a platform should handle tasks like model versioning, monitoring, scaling, and updating to ensure high performance and reliability. Various serving platforms for AI models have been proposed to facilitate deployment [4, 8, 13, 24, 30]. These systems typically use containers to host machine learning models and handle inference requests via REST or RPC APIs. For instance, Clipper [13] and DLHub [8] adopt separate Docker containers to ensure process isolation, while BentoML [4] offers a unified deployment framework that acts as a bridge between machine learning frameworks and serving platforms, simplifying the process of creating machine learning services that are ready to deploy and scale. Additionally, Pretzel [24] proposed a white box model that optimizes of prediction pipelines with resource sharing. However, the above platforms are not well-suited for AIoT applications, as they utilize a request-response pattern and may fail to accommodate heterogeneous data processing in AIoT applications.

3 DESIGN OF AIOT PLATFORM

In this section, we present the architecture of our platform, as shown in Fig. 1. There are four main components: *Device*, *Service*, *Central Broker*, and *AI Model*.

The *Device* component is primarily composed of various sensors. These devices can collect and transmit data to the *Service* component. Each device may have multiple sensors that collect different types of data from the environment. These data can range from basic measurements such as temperature and humidity to more complex information such as images and voices, depending on the sensor types. Once data is collected, a device can transmit it to the *Service* component for further processing and analysis. In addition, the devices exhibit versatility in receiving processed results. Unlike traditional systems, where devices are constrained to receive their own processed data, our platform enables a device to receive computed outcomes from other devices. This cross-device communication enhances the system's interconnectedness and allows for a broader scope of potential applications among various devices.

The *Service* component serves as an interface between the Device and AI Model components. By preventing direct interactions between a device and an AI model, such a separation of layers ensures that the complexity of AI models is abstracted away from devices, establishing a controlled environment for data handling and processing. This component provides a set of RESTful APIs to the application developers to manage services effectively, allowing for querying, adding, and deleting existing services as needed. Each service within this component is designed to connect with a variety of AI models. This capability provides the flexibility for services to offer complex and customized functionalities according to different needs. Notably, the *Service* component is designed in such a way that each service operates independently. The independence of services allows for the seamless addition and deletion of services without impacting others, promoting system robustness and scalability. Moreover, each service is designed with its independent message broker to facilitate the collection and transmission of data, enabling efficient and reliable data flow.

The *Central Broker* leverages RabbitMQ's message broker [32] capabilities and serves as the communication hub. Each service and AI model can effectively exchange data through this message broker. The broker utilizes exchange bindings with different AI models to orchestrate complex inferencing processes. Each AI model can have its own exchange binding, providing the system with a powerful mechanism for managing message routing based on the service configuration's specific needs and capabilities. This ability to customize message routing to different AI models allows for intricate decision-making and inferencing processes, thus substantially contributing to the platform's ability to handle complex AIoT applications. Another important feature of the RabbitMQ Central Broker is its ability to support distributed computing through its queue management. Multiple instances of the same model can share a queue, allowing for the concurrent processing of messages. This shared-queue design enables the system to distribute computational loads across multiple instances of the same AI model, enhancing the platform's scalability and efficiency, particularly in high computational demands scenarios.

The *AI Model* component operates different pre-trained AI models to consume the data generated by the *Device* component. As in previous work [8, 13], these models are encapsulated as lightweight containers following a microservice architecture to simplify the deployment process. This encapsulation process promotes a standardized and simplified workflow, enabling easy model deployment and allowing developers to focus on model development rather than the intricacies of deployment. It also enhances the portability of AI models, as they can be effortlessly moved or replicated across different environments. Management of these containers is accomplished through a Docker Swarm cluster [20], a native clustering and scheduling tool for Docker. Docker Swarm manages, deploys, and scales these encapsulated AI models across the platform. It provides a reliable and automated way of handling the lifecycle of these model containers, ensuring their effective distribution and operation within the platform. With Docker Swarm, users can scale up the system based on computational demands. It can add model replicas as needed, allowing for a dynamic allocation of resources that can handle varying workloads. This capability is crucial for maintaining system performance during peak-usage as well as under-usage periods, thus ensuring the platform to remain responsive and effective in resource allocation.

3.1 AMQP Communication Architecture

The communication architecture of our AIoT platform is supported by the AMQP broker [40], an open standard application

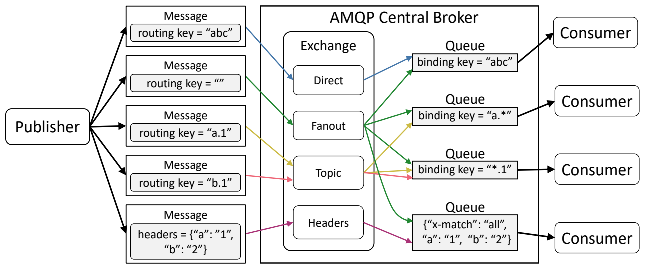

Figure 2: AQMP broker architecture.

layer protocol for message-oriented middleware. The AMQP broker essentially aids in facilitating message transmissions between connected devices, services, and AI models. Its interoperability, reliability, and standardization make AMQP a favorable choice for our AIoT platform.

The AMQP broker architecture is illustrated in Fig. 2. To effectively route messages based on their binding keys, there are four fundamental exchange types: Direct, Fanout, Topic, and Headers. Each type of exchange has its distinct routing capabilities to facilitate message exchange within the system. In direct exchange, messages are routed to queues with a binding key that exactly matches the routing key of the messages. This allows for direct one-to-one communication between a sender and a receiver. In fanout exchange, a message is broadcast to all the bound queues of the system, thus enabling distributing a message needs to multiple receivers simultaneously. Topic exchange allows for more complex routing based on pattern matching. A message can be routed to one or many queues based on partial match between the routing key and the patterns specified by queues. The special character "*" matches exactly one keyword, while "#" can match any number of keywords, including none at all. Keywords are separated by a period, denoted as ".". In header exchange, routing decisions are determined by message header attributes. This provides an alternative for routing in more complex scenarios where key-based routing might not be sufficient. When certain header attributes meet a specific "x-match-expression," a message will be sent to a queue, where "x-match" could be in the form of *all* or *any*. For example, in Fig. 2, a message with a header attribute *{"a":"1", "b":"2"}* will be routed to the queue with binding key = *{"x-match":"all", "a":"1", "b":"2"}*, while a message with a header attribute *{"a":"1"}* will not pass the *all* criteria.

Another remarkable feature of AMQP is the "Exchange to Exchange Binding," which allows an exchange to be bound to another exchange. That is, an exchange can be further bound to an exchange, thus creating a more complex routing scenario. Therefore, a message can traverse multiple exchanges before reaching its destination queue(s). This supports more dynamic routing to meet the diverse AIoT application needs. For example, consider a scenario where the topic exchange in Fig. 2 serves as a source exchange and is bound to the fanout exchange as the destination exchange. This binding facilitates a message with a routing key *b.1* to be routed first to the topic exchange, passed to the queue with binding key = "*.1", and then relayed to the fanout exchange. Therefore, all queues bound to this fanout exchange will receive the message.

The choice of the ways to exchange data is particularly crucial in future AIoT applications, where devices and data types are diversified and data should be routed to appropriate AI models for processing. We raise some examples below. The first example demonstrates that topic exchange facilitates routing messages to one or multiple queues based on a partial matching rule to ensure that data from specific types of devices can be channeled to the AI models designed to process that data. Imagine a scenario where we have several cameras and accelerometers in a smart healthcare AIoT setting. The AI models designed to interpret camera images and acceleration data are different. A routing key "sensor.camera" can be specified for routing image data, and another "sensor.acc" for routing acceleration data. The corresponding AI models can subscribe to these keys for proper data processing.

The second example demonstrates the capability of using "exchange to exchange binding" to enhance topic exchange. It allows exchanges to be connected as a chain or a graph by routing messages through multiple exchanges. Imagine the face recognition task in gate control, where a chain of AI models, including object detection, fake face detection, and face/ID recognition, need to be executed sequentially. In addition, in case of a fake face being detected, a tracking model may be triggered to find the roaming paths for potential intruders. We could design a set AI models connected as a graph via exchange to exchange binding to achieve this goal.

3.2 Chaining of Modules

Based on the exchange mechanisms of AMQP, we chain the Device, Service, and AI Model components in our architecture. To better illustrate the functionality and flow of our AIoT platform, let us consider a scenario in Fig. 3, which involves two devices (Device1 and Device2), two AI models (Model1 and Model2), and two services (ServiceA and ServiceB). Both Device1 and Device2 send out image data, but their data are processed differently by ServiceA and ServiceB, respectively.

Model1 is an object detection model, which takes an image as input and outputs the identified objects in the image. Model2, on the other hand, is a pose estimation model that takes an image as input and outputs text data describing the joints in the pose(s). ServiceA and ServiceB are two distinct services that leverage these AI models. ServiceA only employs Model1, and hence, only performs object detection. In contrast, ServiceB uses both Model1 and Model2 by executing object detection followed by pose estimation.

In this configuration, Device1 connects to the ServiceA broker. The image data from Device1 is sent to the topic exchange within ServiceA broker with routing key = *ServiceA.Device1.InputQueue*. The data is put in *InputQueue*. ServiceA process uses binding key = *ServiceA.*.InputQueue* to receive the above data and forwards it to Model1 using routing key = *ServiceA.Device1.null.image*. Device2, on the other hand, connects to ServiceB broker by sending its data using routing key = *ServiceB.Device2.InputQueue*. ServiceB process then transmits this data using routing key = *ServiceB.Device2.null.image*. Model1 uses binding key = *.*.*.image to get image data from both ServiceA and ServiceB. Model1 conducts inference on each input image and performs object detection. After the inference is done, Model1 sends out its output to

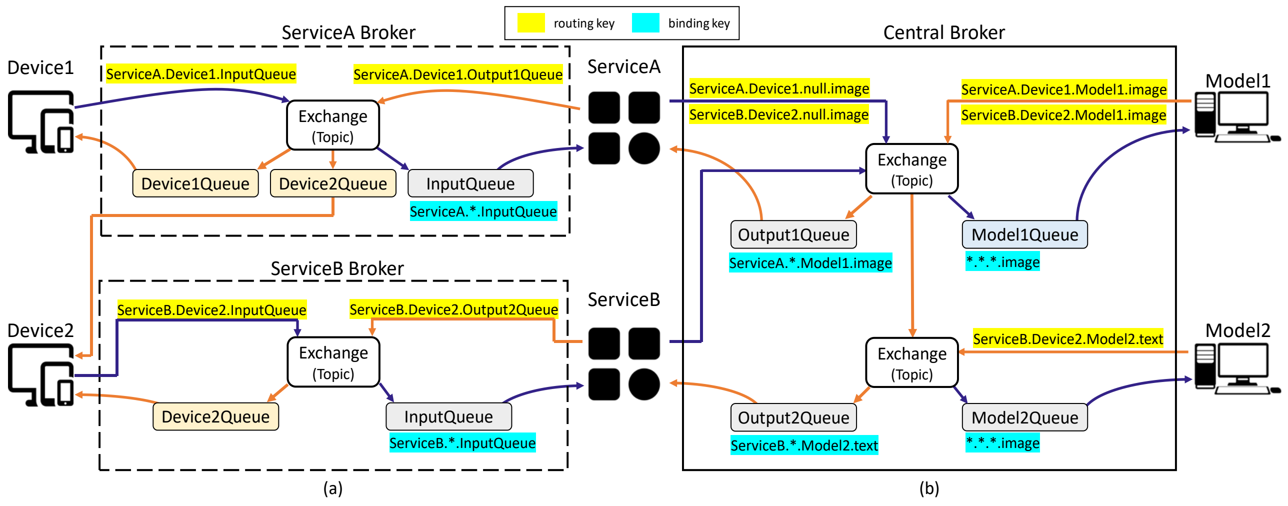

Figure 3: Examples of module chaining: (a) Device to Service and (b) Service to AI Model.

Device1 or Device2 using *ServiceA.Device1.Model1.image* or *ServiceB.Device2.Model1.image* as routing key, respectively. In the case of ServiceB, the exchange-to-exchange binding will use binding key = **.*.Model1.image* to ensures that the data processed by Model1 from Device2 is sent to Model2 for further processing. The final pose estimation result from Model2 is dispatched using routing key = *ServiceB.Device2.Model2.text*. Lastly, ServiceA and ServiceB declare Output1Queue with binding key = *ServiceA.*.Model1.image* and Output2Queue with binding key = *ServiceB.*.Model2.text*, respectively, in the central broker to collect the results from Model1 and Model2. These results are then sent back to their corresponding devices with routing keys = *ServiceA.Device1.Output1Queue* and *ServiceB.Device2.Output2Queue*, respectively.

3.3 AI Model Deployment

Next, we introduce the details to deploy an AI model (e.g., Model1 and Model2 in Fig. 3) within our AIoT platform. We use deploying an object detection model as an example. The code in Listing Fig. 4 shows the encapsulation of an object detection model class for consuming messages, executing inference, and publishing results to the central broker. Such a design offers a high degree of flexibility by allowing models of the same inputs and outputs but different implementations to use the same exchange and queue, enabling easy updates to AI models without the need to redesign the communication architecture. This allows for simple version updates of AI models without the necessity of reconfiguring the communication framework. For instance, one can easily switch between FastRCNN [17] and YOLO [33] for an object detection task, where a model developer only needs to modify the inference code to deploy an alternative model.

Upon instantiation, the Model class initiates a blocking connection to the central broker and establishes a communication channel. It then declares a topic exchange with its ModelName and 'arguments' that outline the outputs of the model. Concurrently, a queue is declared and bound to the exchange using a binding key. This key, formatted as *Service.Client.Model.Datatype*, enables the queue to accept data from any client across different services and even data processed by other models, while the *Datatype* keyword of the key designates the queue's specific task type. For example, an object detection model may be designed to accept either an original PNG image or an already converted RGB formatted tensor. Depending on the Datatype received, the model would perform different operations. This flexibly handles different data types, greatly enhancing the adaptability and functionality of the model.

When the computed results are published to the model's own exchange, the routing key follows the same *Service.Client.Model.Datatype* format. However, in this case, the *Model* keyword is replaced with the ModelName of this specific model and the *Datatype* keyword is modified according to the format of the computation results. With the exchange-to-exchange binding feature, the results can be further routed to the exchange of another model, facilitating a chain of model operations and enhancing the versatility and complexity of the system's tasks.

Moreover, by merging this design with containerization, it is possible to share exchange and queue across multiple instances of the same model. This distribution of computational tasks significantly boosts the system's overall computational capacity. The design thus bridges the gap between message consumption, model computation, and result distribution, maximizing system efficiency and offering a higher level of performance.

3.4 Service Configuration

Our platform can accept and process a service configuration file that describes the service workflow. It enables a dynamic and flexible way for application developers to define their services based on specific application requirements.

To facilitate this, application developers upload a configuration JSON file through the API provided by the Service Controller in the *Service* component. This file describes the service workflow and thus can support a wide range of use cases. Upon receiving this service configuration JSON file, the Service Controller within the

```
class Model():
  def __init__(self) -> None:
    self.connection = BlockingConnection(host, port)
    self.channel = self.connection.channel()

    self.exchange_declare(
      ModelName="YOLO",
      "topic",
      arguments=["image", "text"])
    self.queue_declare(ModelQueue="image")
    self.queue_bind(
      ModelQueue="image",
      ModelName="YOLO",
      binding_key)

    self.channel.basic_consume(
      ModelQueue="image",
      self.__callback)
    self.channel.start_consuming()

  def __callback(self, data) -> None:
    # Model inference here
    result = Model(data)
    self.channel.basic_publish(
      ModelName="YOLO",
      routing_key,
      result)
```

Figure 4: Python code for deploying an object detection model (by YOLO).

platform first validates the content in the file. It verifies that its format meets the specification. This phase is crucial as it protects the platform from possible erroneous or incompatible configurations, thereby maintaining the overall system integrity. After the validation, the Service Controller parses the JSON file, creates a service process, and then creates corresponding queues in the service broker and central broker (refer to the example in Fig. 3). These queues serve as pipes for inter-process communications and data transfer, streamlining the operation and coordination between devices and AI models. Furthermore, to facilitate the incorporation of AI models, the service process sets up exchange-to-exchange bindings in the Central Broker for AI models in the service configuration. This function serves as communication junctions among AI models. The service descriptions in the JSON file set up a chain of model operations in our system.

In Listing Fig. 5, we outline a configuration for a “VisionFallDetector” service, a vision-based fall detection system. The different sections of the JSON file are explained as follows:

service: This field describes the service’s name, i.e., "VisionFallDetector." The Service Controller uses this to create a standalone service broker and service process. The service process establishes a topic exchange within the service broker to let devices send input data. This functionality enables devices to access the backend AI models.

central_broker: This defines the host and port of the central broker. This facilitates connecting the service and AI models for data exchange.

service_broker: The host and port of this service’s dedicated AMQP broker are specified in this field. Typically, port 5672 represents the connection port for the AMQP connection. Users may specify different ports to represent the connection port used by the service broker. Each service has its independent message broker, which helps manage and distribute traffic efficiently.

input: This field identifies which InputQueue(s) will be created within the service broker to receive and forward device input data to the appropriate backend AI model’s exchange. The input name follows the format *DataName_DataFormat*, and the service process will receive this InputQueue’s data with a binding key following the *Service.Client.InputQueue* pattern. For example, the service process accepts PNG images transmitted from devices through the *png_image* queue. The service process then forwards the data from this queue to the corresponding AI model exchange with the *VisionFallDetector.Client.null.image* routing key. This format specification allows us to define what kind of data each service can accept, enhancing the modularity of our platform.

output: This field details which OutputQueue(s) will be set up in the central broker to receive the computation results from AI models. These results are then forwarded to the service’s exchange, enabling devices to access the computed data through DeviceQueue. The output name follows the format *DataName_DataFormat*, and the service process uses a routing key *Service.Client.OutputQueue* to deliver the computation results to its exchange within the service broker. For instance, the service process receives the computed fall detection text results from the fall detection model. These results come from the *FallDetection_text* queue and are sent to the service’s exchange using the *VisionFallDetector.Client.FallDetection_text* routing key. This setup allows the reception of partial or all AI model computation results and forwards these to the device for further processing.

routing: This field represents the service’s data flow by source-destination pairs, including data flow from InputQueue(s) to AI model(s), the linkage between different AI models, and the computation results of AI models to OutputQueue. The routing forms a DAG (directed acyclic graph) dataflow graph. In our example, the service process receives a PNG image from devices through the *png_image* queue and then sends it to the HumanDetector model for processing. Once the HumanDetector model processes the image, it passes a human bounding box to the FallDetectorGCN model, which performs human fall detection. The final results are sent to the *FallDetection_text* queue. This approach allows one to establish connections using the deployed AI models and predefined input-output formats. Therefore, intermediary results can be passed between multiple AI models.

In summary, the service configuration has the advantages of easy customization, agility, and scalability. It provides a clear and concise view of the service structure and can adapt to various requirements of AIoT applications.

4 A CASE STUDY IN SMART HEALTHCARE

Smart healthcare holds a broad range of applications in AIoT, from monitoring, examination, and surgery to rehabilitation [43]. Below, we conduct a case study in healthcare-related accident monitoring for elders with constrained mobility. In particular, fall detection [41] is studied.

```
{"service": "VisionFallDetector",
 "central_broker": {
   "host": "HostName",
   "port" 5672
 },
 "service_broker:" {
   "host": "HostName",
   "port" 5672
 }
 "input": ["png_image"],
 "output": ["FallDetection_text"],
 "routing": [
  {"source": {"type": "input", "queue": "png_image"},
   "destination": [
     {"type": "server", "queue": "HumanDetector_input_image"}
   ]},
  {"source": {"type": "server", "queue": "HumanDetector_output_image"},
   "destination": [
     {"type": "server", "queue": "FallDetectorGCN_input_image"}
   ]},
  {"source": {"type": "server", "queue": "FallDetectorGCN_output_text"},
   "destination": [
     {"type": "output", "queue": "FallDetection_text"}
   ]}
 ]
}
```

Figure 5: Service configuration for a vision-based fall detection application.

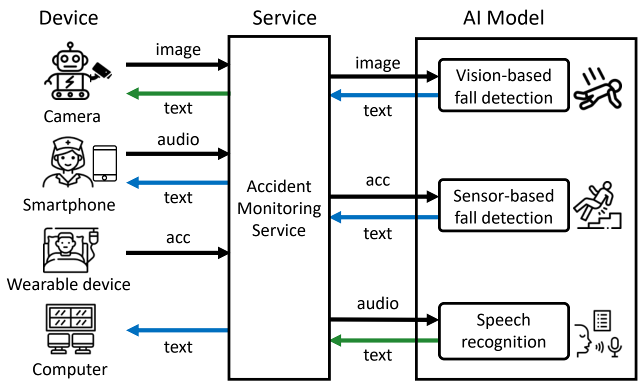

Figure 6: Case study of a smart healthcare application.

4.1 Service Specification

The specification of the accident monitoring service is defined in Fig. 6. It employs a speech recognition model, allowing nursing staff to control robots via voice commands remotely. These robots use a vision-based model to detect falls in real-time from video feeds, while patients are equipped with wearable devices using a sensor-based model for fall detection.

Three AI models are to be deployed in this service. The first model is a vision-based fall detection model as shown in Fig. 7(a), which accepts image inputs and outputs fall detection result in a text form. This model uses Tiny-YOLO [33] to identify people in a frame and crops the bounding boxes with humans. AlphaPose [15] is then used to obtain the skeleton of each person, followed by ST-GCN [42] to predict each person's action in a duration of 30 frames (in skeleton forms). If the predicted action is Fall or Lying down, the model will output a warning message to its exchange.

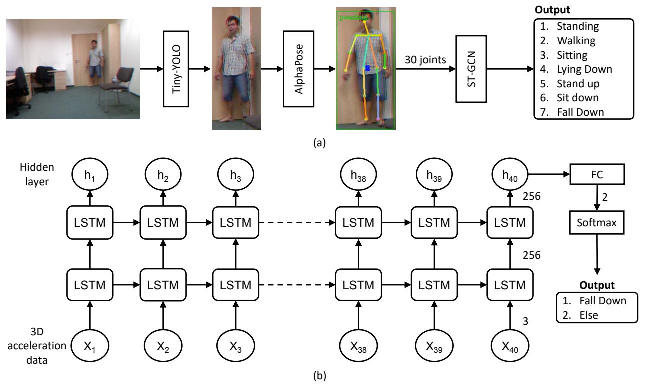

Figure 7: Architecture of (a) vision-based and (b) sensor-based fall detection model.

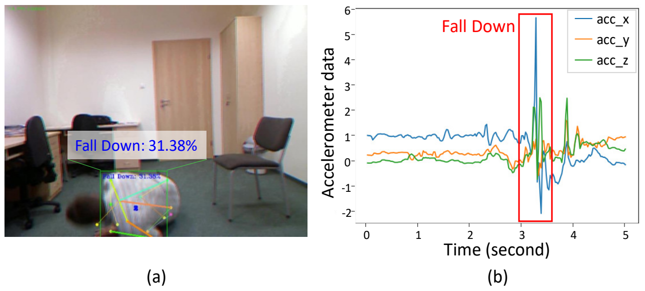

Figure 8: Visualization results of fall detection: (a) vision-based and (b) sensor-based.

The second AI model is for fall detection but in a sensor-based manner, as shown in Fig. 7(b). It receives 40 records of 3D acceleration data and determines whether a fall has occurred. To achieve real-time detection, we adopt a lightweight double-layer Long Short-Term Memory (LSTM) network. In other words, as soon as a potential fall is detected, the model can instantly provide an alert or intervention. Some visualization results of the vision-based model (on the UR fall detection dataset [23]) and the sensor-based model (on the Smartwatch dataset [27]) are presented in Fig. 8.

The third model is a speech recognition model. This model is used to receive a voice command and transform it into a text form. Through text commands, a robot in this service can be remotely controlled in a hand-free manner. We adopt the Deep Speech framework [19], which can ensure accurate, real-time voice-to-text translation, allowing for swift responses to potential emergencies.

4.2 Implementation Details

To deploy this service on our platform, we derive the service configuration file in Fig. 9. Four devices are involved: (i) cameras equipped by robots, smartphones equipped by nurses, (iii) wearable devices equipped by patients, and (iv) a computer in the monitoring center. A nurse uses a smartphone to give voice commands to operate a robot. A voice command is transmitted to the accident exchange in the service broker with a routing key of *Accident.Nurse.wav_audio*. The audio is then forwarded to the

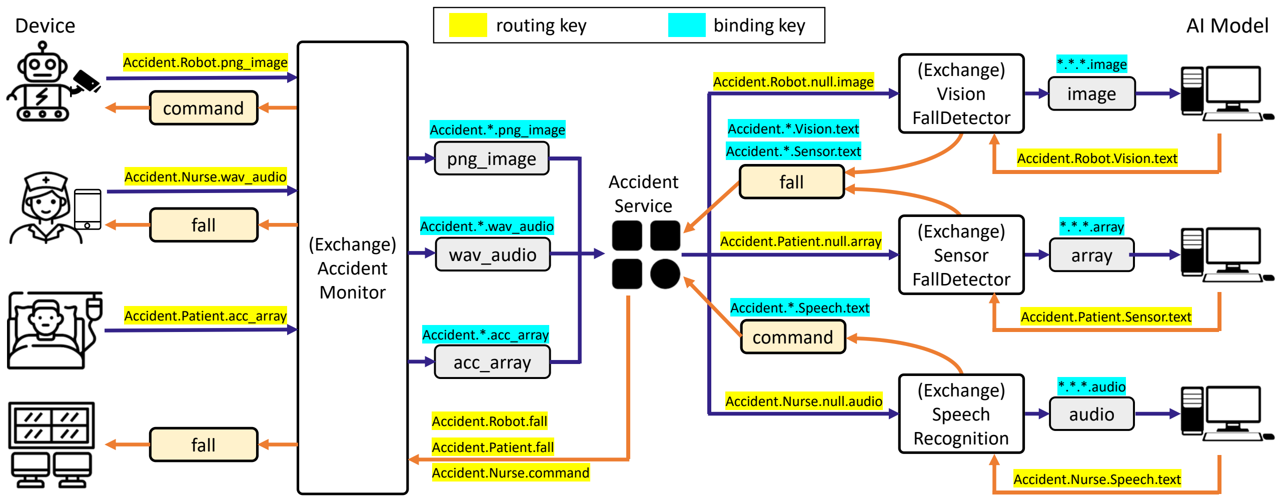

Figure 9: Dataflow of the smart healthcare example.

speech recognition model. The resulting text command is sent to the command queue in the central broker with a routing key of *Accident.Nurse.Speech.text*. This then enables the service process to transfer it to the command queue declared by the robot in the service broker. After receiving the command from the queue, the robot activates its camera and starts its patrol task, continuously sending real-time images to the accident exchange with a routing key of *Accident.Robot.png_image*. Note that this design supports multiple robots to perform patrolling tasks at the same time. Any idle robot receiving a request from the command queue can immediately start its task. The vision-based fall detection model in the service uses images sent by robots to detect falls. If a fall is detected, a message is sent to the central broker's fall queue with a routing key of *Accident.Robot.Vision.text*. This message is then forwarded to the fall queue of the nurse and monitoring center in the service broker. Simultaneously, the patient's wearable device continuously sends accelerometer data collected from the patient, using the routing key *Accident.Patient.acc_array* to the accident exchange. The sensor-based fall detection model uses this information to calculate fall results. These results are then sent to the fall queue with a routing key of *Accident.Patient.Sensor.text*. Thus, the nurse and monitoring center can receive fall alerts detected not only by the robot but also directly from the patient, allowing hospital staff to respond promptly.

This case study demonstrates the distinctive advantages of our Exchange-based AIoT Platform, notably including its extensibility, capacity for intelligent decision-making, and facilitation of fast application development. For extensibility, multiple devices can be integrated concurrently, as well illustrated in the case study where multiple robots, smartphones, wearable devices, and computers are seamlessly linked. For supporting intelligent decision-making, multiple advanced AI models are integrated, such as speech recognition for voice-to-text conversion and two-modality fall detection by visual and wearable sensors. For fast application development, by utilizing a service configuration file to describe the service flow, developers can easily reconfigure their workflows and concentrate on the application logistics. This would accelerate AIoT application development processes.

5 COMPARISONS AND EVALUATIONS

5.1 Functionality Comparisons

In this section, we compare our platform with Node-RED and BentoML.

Node-RED [11] is an open-source visual programming tool that provides a browser-based flow editor. It lets users connect and configure nodes visually to create applications and automate workflows. Node-RED follows a 'flow-based programming‘ approach, where nodes represent different functionalities or services and are connected to define the logic and data flow of the application. However, Node-RED's limitation in supporting diverse deep learning frameworks, such as PyTorch, Keras, or Caffe, poses a drawback when developers seek to employ models built with these frameworks. In contrast, our platform's containerization capability enables seamless integration of multiple deep learning frameworks, offering greater flexibility in model utilization. This flexibility allows developers to leverage a wide range of pre-existing models and easily switch between implementations without necessitating modifications to the communication architecture. Moreover, our platform's approach eliminates the need for communication framework reconfiguration when updating AI models. The process of model updates and version control is simplified. In contrast, Node-RED requires potential reworking of the communication architecture when handling model updates and versioning.

BentoML [4] is a high-level, Python-based model serving framework designed to bridge the gap between Data Science and DevOps. It adopts a Service-Oriented Architecture (SOA) approach to define services, employs 'Runners' to denote the computational logic of models, allows API definition for specifying service input and output, and supports the use of multiple models for complex inference tasks. Fig. 10 shows how to derive our smart healthcare case study in BentoML. For analogy, API1, API2, and API3 correspond to our

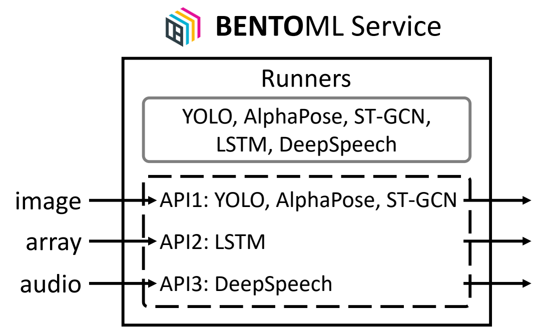

Figure 10: Comparison of BentoML to our platform in the smart hearlthcare case study.

vision-based fall detection model, sensor-based fall detection model, and speech recognition model, respectively.

Our AIoT platform shares some similar features with BentoML, but stands out through several distinctive features. First, both platforms use multiple server queues for a model to operate with different inputs (the 'Runner' function of BentoML and the service input and output queues of ours). However, our platform supports an additional advanced exchange-to-exchange binding mechanism. Second, unlike BentoML, which operates solely on a single machine, our platform can distribute models across multiple machines for decentralized processing. For instance, our smart healthcare application's vision-based fall detection model incorporates YOLO, AlphaPose, and ST-GCN models, each running on a different machine. Third, BentoML uses a request-response (one-to-one) pattern, while our platform supports both request-response and publish-subscribe patterns to enable multipoint (many-to-many) communication. For example, in our smart healthcare application, through simple configurations, the fall detection results derived from both vision-based and sensor-based models are sent to both nurses and the monitoring center. Fourth, BentoML's API only returns the final computational result and thus provides no insight into intermediate outcomes. Our platform allows access to intermediate results during a sequence of computational processes, thus improving the explainability of a system. For example, in our smart healthcare application, the human pose structure generated by AlphaPose can be sent back to the monitoring center. Recording the pose structure during a fall may help doctor's diagnosis.

In essence, our AIoT platform introduces a new level of versatility and adaptability when compared to traditional models like BentoML. With support for distributed processing, diverse communication patterns, and flexible result handling, it emerges as an innovative tool for AIoT applications.

5.2 Performance Evaluation

To understand how our AIoT platform performs, we test the vision-based fall detection model by focusing on the design of YOLO and AlphaPose components. We decouple them into separate containers and feed a total of 600 images at a rate of 100 images per second for 6 seconds. We evaluate the serving time to process all images. The experiment uses two machines: Machine1 with an Intel Xeon Silver 4210R CPU and 256GB of RAM, and Machine2 with an Intel i7-12700k CPU and 128GB of RAM. Both Machine1

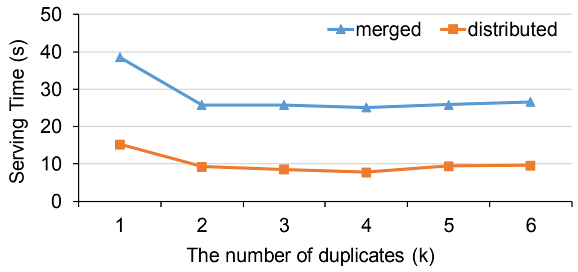

Figure 11: Comparison on using merged and distributed containers for YOLO and AlphaPose.

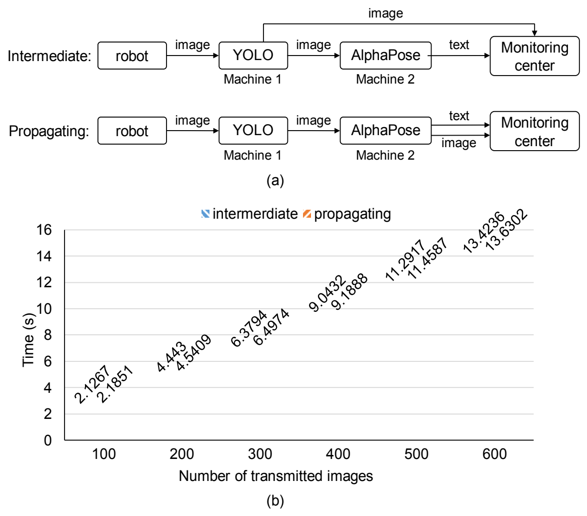

Figure 12: Comparison on 'intermediate' and 'propagating' dataflows.

and Machine2 use an NVIDIA RTX 3090 GPU for inference. We consider two scenarios: (i) merged (putting YOLO and AlphaPose containers on Machine1), and (ii) distributed (YOLO container on Machine2 and AlphaPose container on Machine1).

In both tests, we duplicate each container k times, where $k = 1..6$, to evaluate scalability under different loads. As shown in Fig. 11, the distributed setup consistently outperforms the merged setup across all ks. It also shows that increasing k can reduce service time in the beginning, but would lower down performance afterwards. The increase in service time after $k \geq 5$ is likely due to RabbitMQ's round-robin message distribution mechanism and resource contention.

We also evaluate the performance difference between (i) sending intermediate results directly to the monitoring center, and (ii) propagating intermediate results step-by-step until reaching to the monitoring center. Fig. 12(a) illustrates the two dataflows. In Fig. 12(b), we evaluate the time taken for the monitoring center to receive all results by varying the number of images transmitted. The 'intermediate' setup consistently takes less time than the 'propagating' setup in all cases. Moreover, the time difference enlarges as more images are transmitted. This outcome demonstrates the advantages of allowing sending out intermediate results in advance.

6 CONCLUSIONS

In this work, we point out some limitations of existing IoT and AI model serving platforms, particularly regarding interoperability and extensibility. Our AIoT platform, which uses microservice architecture and AMQP middleware, can effectively address these problems. It simplifies building complex inference tasks and allows for integrating various types of AI models. Through a smart healthcare application, we demonstrated how our platform relieves the above limitations by exploiting many-to-many communications. Further, we showcased the advantages of our platform's modular design by evaluating the performance of distributed computing and intermediate result dispatch mechanism. Overall, our platform offers a flexible foundation for AIoT applications, demonstrating the potential to transform the AIoT development process and ecosystem.

ACKNOWLEDGMENTS

Y.-C. Tseng's research is co-sponsored by ITRI and NSTC, Taiwan. This work is also financially supported by "Center for Open Intelligent Connectivity" of "Higher Education Sprout Project" of NYCU and MOE, Taiwan. The research of Yi Ren was supported in part by EPSRC EP/T022566/1, EP/T024593/1, and the Royal Society IEC\R3\213100.

REFERENCES

[1] Eyhab Al-Masri, Karan Raj Kalyanam, John Batts, Jonathan Kim, Sharanjit Singh, Tammy Vo, and Charlotte Yan. 2020. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access* 8 (2020), 94880–94911.

[2] Farman Ali, Shaker El-Sappagh, SM Riazul Islam, Daehan Kwak, Amjad Ali, Muhammad Imran, and Kyung-Sup Kwak. 2020. A Smart Healthcare Monitoring System for Heart Disease Prediction Based on Ensemble Deep Learning and Feature Fusion. *Information Fusion* 63 (2020), 208–222.

[3] Karen Avila, Paul Sanmartin, Daladier Jabba, and Miguel Jimeno. 2017. Applications Based on Service-Oriented Architecture (SOA) in the Field of Home Healthcare. *Sensors* 17, 8 (2017), 1703.

[4] bentoml.com. 2019. BentoML. https://www.bentoml.ai/.

[5] Björn Butzin, Frank Golatowski, and Dirk Timmermann. 2016. Microservices Approach for the Internet of Things. In *Proc. IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–6.

[6] Gustavo Caiza, Erick S Llamuca, Carlos A Garcia, Fabian Gallardo-Cardenas, David Lanas, and Marcelo V Garcia. 2019. Industrial Shop-Floor Integration Based on AMQP protocol in an IoT Environment. In *2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM)*. IEEE, 1–6.

[7] Carlos Cambra, Sandra Sendra, Jaime Lloret, and Laura Garcia. 2017. An IoT Service-Oriented System for Agriculture Monitoring. In *Proc. IEEE International Conference on Communications (ICC)*. 1–6.

[8] Ryan Chard, Zhuozhao Li, Kyle Chard, Logan Ward, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J Franklin, and Ian Foster. 2019. DLHub: Model and Data Serving for Science. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 283–292.

[9] Ting-Hui Chiang, Zao-Hung Sun, Huan-Ruei Shiu, Kate Ching-Ju Lin, and Yu-Chee Tseng. 2020. Magnetic Field-Based Localization in Factories Using Neural Network With Robotic Sampling. *IEEE Sensors Journal* 20, 21 (2020), 13110–13118.

[10] Michele Ciavotta, Marino Alge, Silvia Menato, Diego Rovere, and Paolo Pedrazzoli. 2017. A Microservice-Based Middleware for the Digital Factory. *Procedia manufacturing* 11 (2017), 931–938.

[11] OpenJS Foundation & Contributors. 2015. Node-RED. https://nodered.org/.

[12] PyTorch Serve Contributors. 2016. Pytorch Serving. https://pytorch.org/serve/.

[13] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proc. USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Vol. 17. 613–627.

[14] Li Da Xu, Wu He, and Shancang Li. 2014. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics* 10, 4 (2014), 2233–2243.

[15] Hao-Shu Fang, Jiefeng Li, Hongyang Tang, Chao Xu, Haoyi Zhu, Yuliang Xiu, Yong-Lu Li, and Cewu Lu. 2022. AlphaPose: Whole-Body Regional Multi-Person Pose Estimation and Tracking in Real-Time. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[16] Luca Foschini, Tarik Taleb, Antonio Corradi, and Dario Bottazzi. 2011. M2M-based Metropolitan Platform for IMS-Enabled Road Traffic Management in IoT. *IEEE Communications Magazine* 49, 11 (2011), 50–57.

[17] Ross Girshick. 2015. Fast R-CNN. In *Proc. IEEE International Conference on Computer Vision*. 1440–1448.

[18] Google. 2017. Google Cloud IoT. https://cloud.google.com/iot-core.

[19] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep Speech: Scaling Up End-to-End Speech Recognition. *arXiv:1412.5567* (2014).

[20] Docker Inc. 2017. Docker Swarm. https://docs.docker.com/engine/swarm/.

[21] Ruimin Ke, Zhibin Li, Jinjun Tang, Zewen Pan, and Yinhai Wang. 2019. Real-Time Traffic Flow Parameter Estimation From UAV Video Based on Ensemble Classifier and Optical Flow. *IEEE Transactions on Intelligent Transportation Systems* 20, 1 (2019), 54–64.

[22] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti. 2015. Designing a Smart City Internet of Things Platform with Microservice Architecture. In *Proc. International Conference on Future Internet of Things and Cloud*. 25–30.

[23] Bogdan Kwolek and Michal Kepski. 2014. Human Fall Detection on Embedded Platform Using Depth Maps and Wireless Accelerometer. *Computer Methods and Programs in Biomedicine* 117, 3 (2014), 489–501.

[24] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 18. 611–626.

[25] Liangzhi Li, Kaoru Ota, and Mianxiong Dong. 2018. Deep Learning for Smart Industry: Efficient Manufacture Inspection System With Fog Computing. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4665–4673.

[26] Yu-Ting Liu, Jen-Jee Chen, Yu-Chee Tseng, and Frank Y. Li. 2022. An Auto-Encoder Multitask LSTM Model for Boundary Localization. *IEEE Sensors Journal* 22, 11 (2022), 10940–10953.

[27] Taylor R Mauldin, Marc E Canby, Vangelis Metsis, Anne HH Ngu, and Coralys Cubero Rivera. 2018. SmartFall: A Smartwatch-Based Fall Detection System Using Deep Learning. *Sensors* 18, 10 (2018), 3363.

[28] Microsoft. 2023. Microsoft Azure IoT. https://azure.microsoft.com/en-us/solutions/iot.

[29] Nitin Naik. 2017. Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. In *IEEE International Systems Engineering Symposium (ISSE)*. 1–7.

[30] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-Serving: Flexible, High-Performance ML Serving. In *Proc. NIPS Workshop on ML Systems*.

[31] Meng-Shiuan Pan, Yen-Ann Chen, Ting-Chou Chien, Yueh-Feng Lee, and Yu-Chee Tseng. 2010. Automatic lighting control system and method. US Patent 7,843,353.

[32] RabbitMQ. 2023. RabbitMQ. https://www.rabbitmq.com/.

[33] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proc. IEEE conference on Computer Vision and Pattern Recognition*. 779–788.

[34] Julia Robles, Cristian Martín, and Manuel Díaz. 2023. OpenTwins: An Open-Source Framework for the Design, Development and Integration of Effective 3D-IoT-AI-powered Digital Twins. *arXiv preprint arXiv:2301.05560* (2023).

[35] Amazon Web Services. 2023. AWS IoT. https://aws.amazon.com/iot/.

[36] Jeddou Sidna, Baina Amine, Najid Abdallah, and Hassan El Alami. 2020. Analysis and Evaluation of Communication Protocols for IoT Applications. In *Proc. International Conference on Intelligent Systems: Theories and Applications*. 1–6.

[37] Sergio Trilles, Alberto González-Pérez, and Joaquín Huerta. 2020. An IoT Platform Based on Microservices and Serverless Paradigms for Smart Farming Purposes. *Sensors* 20, 8 (2020).

[38] Yu-Yun Tseng, Po-Min Hsu, Jen-Jee Chen, and Yu-Chee Tseng. 2020. Computer Vision-Assisted Instant Alerts in 5G. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*.

[39] Lan-Da Van, Ling-Yan Zhang, Chun-Hao Chang, Kit-Lun Tong, Kun-Ru Wu, and Yu-Chee Tseng. 2021. Things in the air: tagging wearable IoT information on drone videos. *Discover Internet Things* 1, 1 (2021).

[40] Steve Vinoski. 2006. Advanced Message Queuing Protocol. *IEEE Internet Computing* 10, 6 (2006), 87–89.

[41] Xueyi Wang, Joshua Ellul, and George Azzopardi. 2020. Elderly Fall Detection Systems: A Literature Survey. *Frontiers in Robotics and AI* 7 (2020), 71.

[42] Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition. In *Proc. AAAI Conference on Artificial Intelligence*, Vol. 32.

[43] Jing Zhang and Dacheng Tao. 2020. Empowering Things With Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things. *IEEE Internet of Things Journal* 8, 10 (2020), 7789–7817.