Enhancing IoT Products through Integrated AI Capabilities: Enabling Seamless AIoT Implementation

Kerem AYTAÇ
Priva BVA, Rotterdam, Netherlands
Marmara University, Istanbul, Turkiye
keremaytac@gmail.com

Ömer KORÇAK
Marmara University
Istanbul, Turkiye
omer.korcak@marmara.edu.tr

Abstract— **In the contemporary landscape of Internet of Things (IoT) products, mere incorporation of IoT functionalities falls short of meeting the evolving consumer expectations. The paradigm has shifted towards more sophisticated approaches, collectively referred to as Artificial Intelligence of Things (AIoT), wherein IoT devices are empowered by the integration of Artificial Intelligence (AI) capabilities. However, transitioning from conceptualization to realization proves to be a formidable challenge. Proficiency in AI is a distinct discipline requiring specialized expertise; lacking such competencies can lead to potential pitfalls such as wrong or unintended or unreliable results. The attractiveness of AI capabilities remains, prompting stakeholders to explore ways to either achieve expertise or enable user-friendly routes, thereby gaining a competitive advantage.. This paper introduces a novel proposition: the seamless integration of black box AI capabilities into IoT products. This integration stands to be universally applicable across diverse IoT products. The solution eliminates the need for data science expertise, while also providing a significant level of flexibility for end-users to enhance AI capabilities as per their needs. This article outlines the foundational concept of the suggested integration and elaborates on its potential to transition traditional IoT products into the AIoT realm.**

I. INTRODUCTION

In the contemporary technological world, we have lots of useful IoT devices serving various purposes, and there are also AI tools bringing value in unique ways. The idea of putting both IoT and AI together into so-called Artificial Intelligence of Things (AIoT) [1] [2] seems promising, as it could offer attractive benefits that customers desire. However, making this combination work is challenging. Especially for IoT companies without a strong data science background, forming a new team for this can be tough and expensive. Even companies that have both IoT and AI products face difficulties in making them work well together. This is because they have different teams handling each type of product, and getting these teams to collaborate can be challenging. Also, their software development lifecycles are quite different, so making them work seamlessly in sync is not easy, especially with agile methods. To mitigate these problems, AI teams often collect data from IoT devices, apply their business logic in a completely separate pipeline, and provide some useful reports with some latencies. In this approach, IoT and AI products will be abstracted from each other.

For the sake of better understanding, let us illustrate through a simple instance. Imagine a world of manufacturing company producing cars that employs an IoT system to monitor various stages of production using sensors and control actuators. This application elevates production efficiency and quality control by overseeing the manufacturing process. Additionally, the company receives monthly reports based on the collected sensor data. These reports incorporate machine learning algorithms to assess potential issues, such as the risk of a robot arm malfunction during the tire-mounting process. This predictive analysis relies on diverse sensor measurements like vibration, temperature, and arm rotation speed to determine maintenance requirements for specific robot arms. However, monthly reports may not adequately address the potential for abrupt robot arm failures. Relying solely on these reports might lead to unexpected breakdowns, as immediate failures can occur without prior indications. To preemptively identify anomalies, agile intervention is essential. This means that timely detection of irregularities is crucial. Failing to adopt an agile approach might result in missed opportunities for addressing impending issues before they escalate. Swift intervention would have facilitated comprehensive maintenance and minimized downtime, ensuring operational continuity. This approach is commonly referred to as "predictive maintenance" [3]. By anticipating issues in advance and promptly addressing them, the incidence of malfunctioning components within the factory is minimized. Consequently, this practice mitigates unanticipated costs and reduces instances of operational interruptions.

An AIoT product is expected to yield immediate outcomes from real-time data, delivering substantial value to consumers. Nonetheless, constructing precise and useful models that yield high accuracy often demands a substantial investment of time, spanning weeks, months, or even years. Moreover, these models necessitate ongoing adjustments and maintenance, which presents potential challenges when integrating them into IoT products, including a requirement of dedicated and hands-on experienced proficiency which is obviously a challenging process.

Considering an alternative perspective, a conventional product is typically designed for universal application, offering a standardized user experience for all customers without bespoke modifications. Conversely, AI models diverge across customers due to disparities in sensor measurements, application domains, and use cases. The impracticability of crafting a uniform model for diverse clientele necessitates a dual approach: an underlying, shared IoT product designed to serve all customers uniformly, and a tailored AI solution that adapts significantly between customers. This presents a paradox: the IoT product alone lacks the appeal to attract a substantial customer base, while the diversity of AI requirements of customers poses challenges in creating a unified AI model.

To resolve this problem, adopting an AI approach characterized by a self-improving, adaptable, and modular black box is prudent. By embedding rudimentary AI capabilities into the IoT product, customers can initiate usage with basic AI

functionality, which evolves over time as data accumulates. This progressive evolution culminates in the system reaching an optimal state where customers can maximize benefits.

This paper delves into the outlined problem through the lens of the aforementioned use case, "Predictive Maintenance", which serves to offer a tangible illustration and enhance comprehension.

A. RELATED WORK

In the literature, there are many useful applications of AIoT. Sipola et al. [4] describe applications of moving AI computation near the IoT data sources in various domains such as security, mobile networks, healthcare, voice and image analysis and associated frameworks. Sun et al. [5] have proposed a resilient AI system integrated atop the Internet of Things (IoT), which significantly enhances sensor accuracy for the precise identification of object grasps within a virtual retail environment. Chen et al. [6] have demonstrated the merge of AI and image recognition technologies into sensor systems embedded within an IoT framework. This integration facilitates improved pest identification mechanisms within agricultural systems. Zhang et al. [7] have introduced a valuable framework tailored for tunnel construction operations. By collecting specific data, this framework trains models designed to predict operational parameters for both the shield and ground response during subsequent phases of construction. This AIoT-based system augments information granularity and automation throughout the construction process, streamlining decision-making processes and minimizing the occurrence of accidents. In another study, Chen S. W. et al. [8] scrutinize recent advancements in combating the COVID-19 pandemic through the symbiotic utilization of IoT and AI. The article delves into comprehensive strategies for addressing the pandemic and explores prospective technological avenues for such endeavors. Mamza [9] has presented an AIoT based system incorporating medical devices, sensors, and web/mobile applications to create a globally accessible medical resource, enabling monitoring of vital signs like heart rate and blood pressure, even in remote areas without nearby hospitals.

In a closely related study, Prado et al. [10] examine the evolving landscape of next-generation embedded ICT systems, which possess the capability to autonomously execute tasks while being interconnected and collaborative. This research underscores the increasing importance of Edge computing in seamlessly integrating artificial intelligence (AI) into our everyday lives, particularly given the growth of the embedded ICT market. Nevertheless, a substantial challenge emerges from the complexity associated with the harmonization of data, algorithms, and tools for deploying tailored AI solutions on embedded devices, which has impeded widespread adoption. To address this challenge, the authors propose a modular AI pipeline designed to simplify the integration process. This pipeline facilitates end-to-end AI product development for embedded devices and comprises four key stages: data ingestion, model training, deployment optimization, and IoT hub integration.

Furthermore, leading companies in the IoT sector, such as Microsoft, offer a set of generic IoT services that can be enhanced through the incorporation of machine learning capabilities, thus transforming them into AIoT products. These AIoT products can be applied across a wide range of domains [11]. Amazon's AWS IoT Greengrass, for instance, provides the ability to perform on-device inference at the edge while also offering cloud-based management. Although it supports a variety of edge devices, it's important to note that this service is currently available only in specific regions and conforms to Amazon's specific format [12]. In addition to enterprise-level AI services, there's an opportunity to shift our focus towards AI services oriented for edge computing, particularly those designed for vision-related tasks. One notable example is Eugene, a suite of machine intelligence services tailored specifically for IoT applications. Eugene encompasses various functionalities including data labeling, model training, deployment optimization, and integration with IoT systems. This work not only showcases the efficient customization of deep neural networks but also introduces a runtime scheduling algorithm to optimize the selection of network depth [13]. However, it is important to recognize that these approaches generally require a prior understanding of machine learning in order to effectively integrate and utilize them within the intended solution.

II. OVERVIEW OF CURRENT IOT PRODUCT

In this section, we will begin by introducing the prevailing and basic design of an IoT product. On top of that, the proposed AI module will be introduced as well. A generic IoT product revolves around both real-time and historical data. Real-time data is particularly valuable for creating dynamic dashboards, facilitating ongoing monitoring, and enabling instant insights. On the other hand, historical data serves as a valuable resource for generating comprehensive reports and supporting data science endeavors. It is important to note that historical data can be classified into different types. Short-term historical data, for instance, offers higher data freshness and is suitable for storage in hot-storage systems. It typically spans up to weekly data intervals. In contrast, long-term historical data possesses lower data freshness but comprises voluminous data, making it conducive for generating numerous reports and conducting meaningful data science analysis. This data type spans various time ranges, from days to years or even decades. While long-term historical data is particularly well-suited for data science activities, it is worth noting that short-term historical data also holds relevance for similar purposes. Although the outcomes may be constrained, they are immediate in nature, circumventing the need for prolonged data accumulation periods.

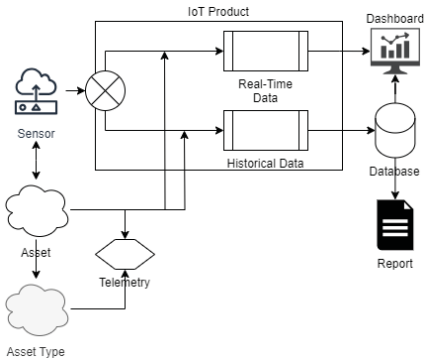

Fig. 1 High-level illustration of a basic IoT product.

Fig. 1 provides a high-level overview of a basic IoT product. A sensor initiates data generation, followed by a multiplexer that duplicates and transmits this data through Real-time and Historical Data channels. Below these channels, a set of services execute the

requisite business logic, and ultimately, the data is directed towards one or multiple databases. Real-time data is typically utilized immediately and discarded once its relevance wanes, whereas Historical data is persistently stored and actively employed for various purposes. In our context, historical data emerges as the optimal source to engage with and utilize. This preference stems from the imperative need for data, which serves as the nourishment vital to training our models. Indeed, data constitutes an essential cornerstone of models; a wealth of data enhances the precision of the models crafted.

In the IoT realm, you also possess entities that are subject to observation or control through sensors and actuators. These entities, referred to as assets, encompass tangible objects within your environment, including components such as a specific model of a robot arm, a section of a factory floor, a door, a production line, or a room. Additionally, asset types serve as foundational templates for assets. For instance, a robot arm could serve as an asset type for all robot arms within a factory. The fundamental attributes of a robot arm, such as vibration and angle sensors, as well as characteristics like model, material, and arm length, represent features inherent to this asset type. Each individual asset emanating from the robot arm asset type must adhere to these attributes. Telemetries like vibration and angle sensors become intrinsic to asset connections, while elements such as model, material, and arm length are stipulated within the system to render them operable. In practical terms, a factory might house numerous robot arms derived from this asset type—possibly numbering in the tens or hundreds. This construct mirrors the concept of an interface or a base class within object-oriented programming. Every piece of data generated within the system is linked to a specific asset. Consequently, it becomes feasible to retrace measurements through the asset's telemetries, simplifying monitoring and management processes. In the IoT context, these entities are commonly referred to as digital twins.

For a more comprehensive examination of the technical dimensions of this product, it is essential to delve deeper. This product is formulated within a containerized environment, specifically residing within Kubernetes [14]. The significance of Kubernetes technology cannot be overemphasized; it indeed stands as a leading-edge solution for hosting applications. While the specifics of Kubernetes are not extensively addressed within this paper, we will harness its preeminent advantages. Within this framework, each application is contained as an individual unit within Kubernetes. For the sake of better understanding, let us consider a scenario involving two applications that collaboratively manage Historical Data. In this scenario, one application acquires data from sensors and refines it, while the other retrieves the refined data and populates multiple databases. These two applications are treated as separate deployments within Kubernetes, operating within distinct boundaries yet residing within the same namespace.

Fig. 2 illustrates the distinct APIs, each functioning as an independent containerized application within the Kubernetes environment. Meanwhile, the utilization of a timeseries database [15] emerges as an optimal choice for the storage of time-dependent data. This database configuration empowers the seamless retrieval of historical data by temporal or aggregated parameters. Conversely, the hot storage database serves as an efficient repository for accessing recent data. Unlike the expansive timeseries database, the hot storage database operates as a cache and is subject to defined data retention policies.

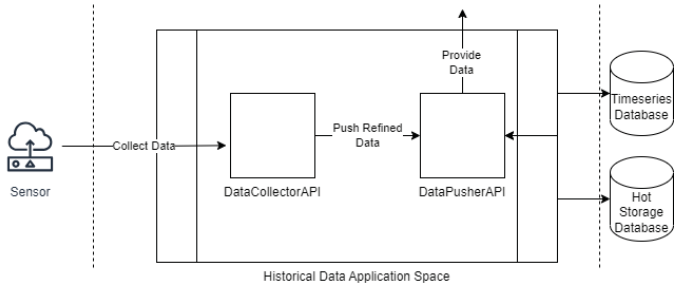

Fig. 2 In-depth exploration of the Historical Data Pipeline to gain detailed insight into the basic IoT design.

In the case of a typical IoT device, let us imagine a customer wants an extra functionality. They want to track the movement of robot arms using sensors. Also, they need a system that can predict when something might go wrong and let the customer know. This project is about warning the customer ahead of time if there might be a problem with the robot arm. This way, the customer knows what's happening and can fix the robot arm before it breaks down completely.

III. PROPOSED SOLUTION

In this section, we provide an elaboration of the proposed solution. Leveraging the benefits conferred by the containerized ecosystem offered by Kubernetes, our approach encompasses the hosting of machine learning models within our system, encapsulated as containers. So, any designated model requires prior containerization as a prerequisite, often executed through the creation of a Docker image [16]. Once a model is dockerized (which can be hosted in a docker environment) [16], its portability is assured, rendering it adaptable to any Kubernetes-supported environment. Typically, data scientists construct their models employing Python [17], a programming language with a diverse array of machine learning libraries. It is important to note that while Python is the conventional choice, other programming languages remain viable alternatives. Within this encapsulated environment, the machine learning process will hide itself into a black box, so that any related parts of this, which is not limited to programming language, but also enabling data wrangling, feature engineering, training, and inference phases of data science will be a minimal concern.

The acquisition of data is an imperative prerequisite for model development. In the context of IoT products, we bear the responsibility of ensuring a continuous stream of data. This sustains the model's capacity to discern patterns, execute data operations, and initiate the process of inference. However, provisioning of the data is not an easy objective. That is, blindly supplying extensive data spanning a year, daily snapshots, or an exhaustive real-time feed encompassing all assets is unsuitable. The parameters governing the extent and frequency of data supply should be defined in accordance with the unique requirements of each model, with the model itself possessing the most astute comprehension of these needs. Within our system architecture, Real-Time and Historical Data applications emerge as the ideal sources for supplying the requisite data. For instance, if a model necessitates data at hourly intervals on a per-minute basis, and its purpose is solely to train on data related to robotic arms, then we should exclusively provide this specific data to maintain efficiency and focus. Furthermore, a model possesses the capability to specify the communication protocol to be employed. This protocol

may encompass familiar standards such as HTTP, AMQP, or alternative protocols. In the case of HTTP communication, the model assumes the responsibility of exposing an endpoint for the purpose of data consumption, and this endpoint should be defined in advance, ensuring a smooth data exchange process.

An IoT product seeks valuable insights just as a model requires data. In this scenario, a connection needs to be established with the model to obtain outputs, which could include an anomaly score and a warning level. These outputs can then be employed for populating dashboards or initiating actions to alert users. It's crucial to define these flexible inputs and outputs in advance. To achieve this, we can create a manifest file once a specific model is developed and utilize it to initiate the model hosting process.

Fig.3 illustrates a representative manifest file, a crucial component in our system.

This file contains a description of the model, the preferred communication protocol, a score endpoint which serves as the destination for collecting data, a health check endpoint which regularly monitor the healthiness of model (the logic for assessing them model's health is embedded within the model itself), hosted location which indicates the real location of model (it is noteworthy that a model can be hosted either within the system or externally, with the latter scenario involving the use of the manifest purely for data collection and proxying to an external model), docker container [18] image information which can be used to pull the model for deployment, trigger type which specifies the type of data the model is interested in (this could be real-time data, where the model receives data as it is generated by the IoT product, or it could be time-window-based, where data is sent periodically with a specified frequency (e.g., per minute) and time window size (e.g., hourly data).), required inputs, and expected outputs that should be clarified as per model. In the given example it will consume temperature data and will return an anomaly score and warning level.

This manifest file possesses human-readable attributes, and it is also capable of being recognized by our system, allowing for seamless configuration. To effectively manage models across various customers, it is imperative to integrate a model management application into the system. This application will have the ability to identify these manifest files, deploy the associated models within the system, establish their operational status, configure the requested data streams, transmit data, monitor the models during operation, assess their health, collect their outputs, and remove models as needed.

Within this application domain, two principal components play a pivotal role, as depicted in Fig. 4. The first component is the Data Companion API, which is created for each model. Its primary responsibility is to collect essential data from the system and feed it into the model as required. Additionally, it directly communicates with the model container to gather outputs, assess the model's health, and perform other related tasks. The second component is the Model Management API, which empowers users to submit manifest files to the system. It guides users in configuring essential settings within the manifest file, such as mapping inputs to telemetries of assets or asset types within the system and mapping outputs to relevant telemetries.

```
{
  "manifest": {
    "description": "Detects anomalies in temperature values",
    "model": {
      "type": "http",
      "scoreEndpoint": "/score",
      "healthCheckEndpoint": "/health",
      "hosted": "internal",
      "container": {
        "image": "temperaturemodel",
        "tag": "latest",
        "customRegistry": {
          "registry": "imageregistry.com",
          "username": "user",
          "password": "password"
        }
      }
    },
    "trigger": {
      "type": "REAL_TIME|TIME_WINDOW",
      "frequency": 5000, //[optional - used for TIME_WINDOW]
      "timeWindowSize": 10000 //[optional - used for TIME_WINDOW]
    },
    "inputs": [
      {
        "dataColumnName": "temperature",
        "readAsString": false
      }
    ],
    "outputs": [
      {
        "dataColumnName": "anomalyScore",
        "readAsString": false
      },
      {
        "dataColumnName": "warningLevel",
        "readAsString": true
      }
    ]
  }
}
```

Fig. 3 An overview of a manifest file for a temperature anomaly detecting model.

Users can also employ this API to halt the execution of models, deploy Model Containers along with Data Companion APIs into the system, and define Data Companion API scopes to collect only the necessary data based on specified conditions in the manifest file. Furthermore, the Model Management API facilitates the collection of insights from the Data Companion API regarding the Model Container. In cases where the model's health is compromised and deemed irrecoverable, the Model Management API has the authority to cease all operations related to that particular model. In essence, the Model Management API serves as the system administrator for model management, while the Data Companion API functions as tightly coupled companion to specific model containers.

The central avenue for our data collection and data output transmission will be the Real-Time and Historical Data Service, which encompasses both the Data Collector API and the Data Pusher API. Through this service, we will gather data in real-time and archive historical data. All outputs generated by our models will serve as telemetry data points associated with specific assets. For instance, the anomaly score telemetry of a robot arm will be enriched with these output data, facilitating their display in user interfaces or triggering predefined actions.

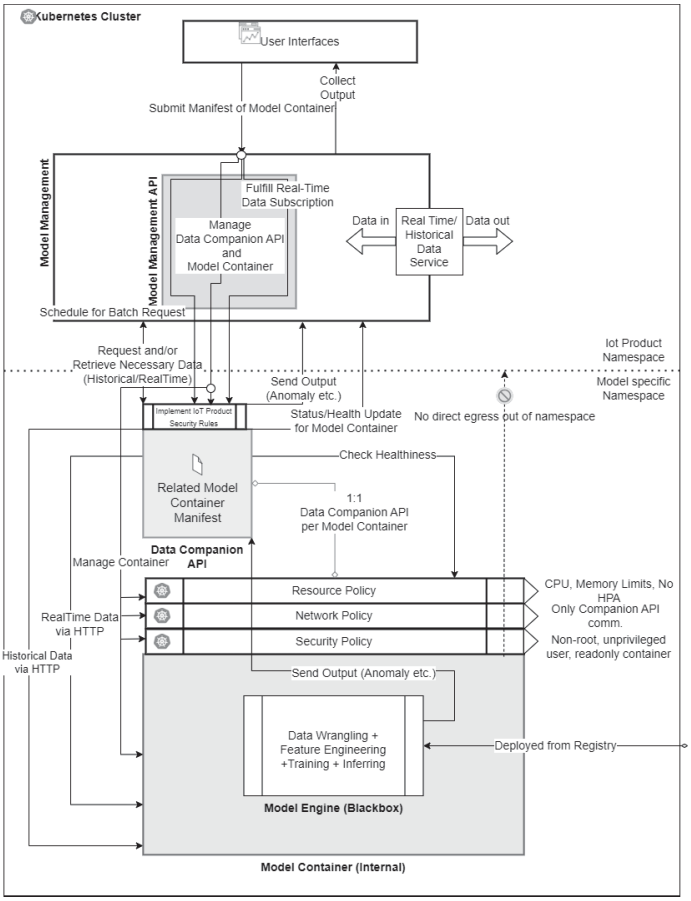

Fig. 4 An overview of Model Management Application with all components and business processes.

In Fig. 4, we observe that the Data Companion API and the Model Container are distinct applications, or Pods, within the Kubernetes environment. These applications are deployed or removed by the Model Management API into separate Kubernetes namespaces, which are also created by this API. Furthermore, the Model Management API is responsible for enforcing certain policies concerning the model containers in alignment with Kubernetes policies. These policies ensure that the created model containers do not excessively consume resources within the Kubernetes cluster. Additionally, they dictate that model containers can only communicate with their respective Data Companion API and must be managed by a non-root, unprivileged user within the container, adhering to well-established Kubernetes security guidelines [19]. The primary motivation behind deploying these models into separate namespaces is to establish isolation between them. This segregation is crucial because our product operates in a multi-tenant IoT environment [20], demanding meticulous attention to security concerns.

The process of managing a model within a Kubernetes system involves various life cycles. The orchestration of these life cycles is overseen by the Model Management API. To achieve an optimal state management, the Saga orchestration [21] pattern is adopted, ensuring that each state triggers corresponding operations. A detailed representation of this process can be found in Fig. 5. Every model adheres to this predetermined life cycle as illustrated in the state diagram, and they persist in an active state unless explicitly removed from the system while in the "draft" phase. Prior to removal, a model must transition into the "draft" state by following the appropriate protocol. In situations where an irreparable health issue arises, the model gracefully reverts to the "draft" state after a series of attempts.

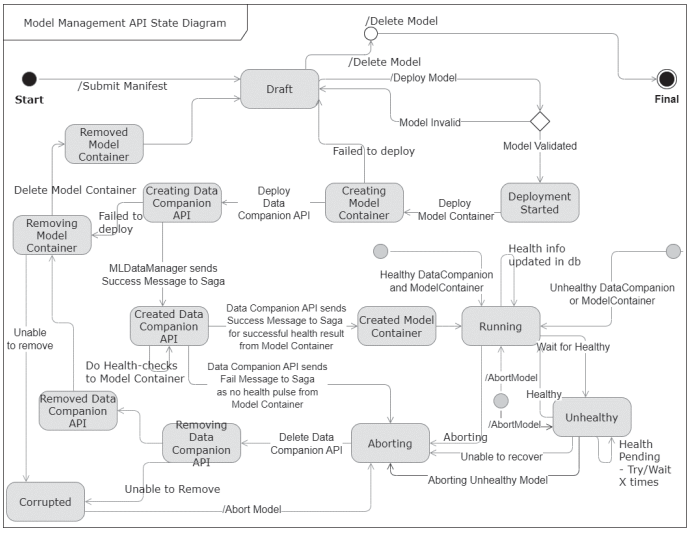

Fig. 5 State diagram of life cycle of a model.

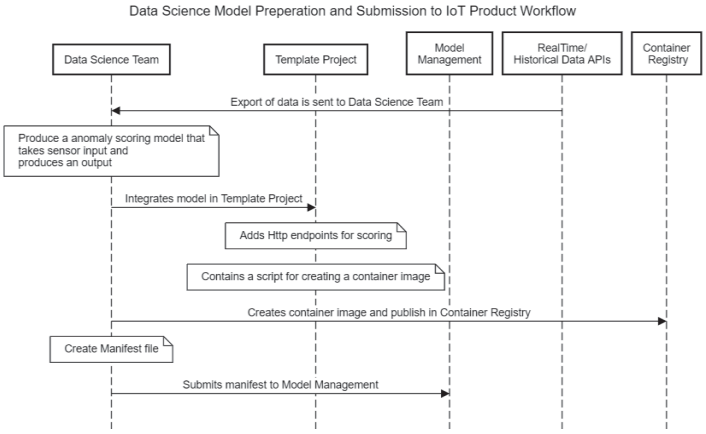

Fig. 6 A fresh start for data science team where they create a useful model.

Firstly, the data science team may require data to conduct testing and validation of their models. To facilitate this, we offer access to our Real-Time and Historical Data APIs, allowing them to acquire the requisite data. Subsequently, the data science team proceeds to craft a high-end anomaly scoring model. They also generate a template project, incorporating essential HTTP endpoints to enable communication with our Data Companion API. This model is then containerized and uploaded to a registry, where it can be later retrieved by the Model Management API for deployment. Additionally, the data science team creates a manifest file in adherence to our predefined structure. This manifest file is prepared for submission into our system, ensuring it complies with our established guidelines and requirements. This procedure can be barely observed as in Fig. 6.

Upon submission of a manifest file and the definition of input and output mappings, this information undergoes interpretation and is subsequently stored within a SQL database. A database schema, as illustrated in Fig. 7, can be adopted for this purpose. In this schema, the "Container Registry" table allows customers to store their container registry details, enabling the system to retrieve their container images. Within the "Model" table, comprehensive information related to the model, sourced from the manifest, is stored. Additionally, vital data pertaining to the model's health and its current state are regularly recorded. The "Model Template Mappings" table encompasses template input and output data, which are initially generated by the data science team. These templates correspond to the input and output sections delineated in the manifest's JSON file. Furthermore, the "Model Mappings" table captures the asset mappings associated with the inputs and outputs specified in the manifest file. For instance, if a manifest file contains "temperature" as an input and "anomaly score" as an output, this table facilitates the mapping of "temperature" to a real "Temperature (Celsius)" telemetry and "Anomaly Score (%)" telemetry for the "Robot Arm X1" asset within our system.

The "AssetTypeMappings" table serves as a critical component when opting to run models against asset types rather than individual assets. In essence, this approach allows for the deployment of a versatile model capable of accommodating any asset falling under a specific asset type, such as "Robot Arm". For example, consider five robot arms denoted as X1, X2, Y1, Z1, and A2, all of which belong to the "Robot Arm" asset type. These robot arms can effectively utilize the same model. When a user establishes a mapping through the asset type, it becomes necessary to internally deconstruct this mapping into individual assets just before deploying the model. This ensures that each asset receives the appropriate model inference. Furthermore, it is crucial to notify the model whenever a new asset is added or removed within a specific asset type. This allows for the seamless inclusion or exclusion of assets in the model's inference process, maintaining its adaptability and accuracy as the asset inventory evolves.

In order to have better understanding of the design, we can refer to Fig. 8. In this diagram, a model is depicted as being stateful, implying that it continuously receives real-time data, gradually undergoing training and acquiring knowledge from this incoming data. It's essential to note that the model's effectiveness increases in proportion to the volume of data it receives. This model becomes fully integrated into our system, establishing itself as an integral component of our operations.

Fig. 9 provides a comprehensive view of the entire state diagram, offering valuable insights into the sequence of states and operations. Within this diagram, we also encounter a stateless model, which operates differently compared to its stateful counterpart. In the case of a stateless model, it retrieves a batch of data and conducts both training and inference operations within the confines of this specific dataset, yielding results accordingly. Subsequent batches of data entirely replace the previous ones, initiating a distinct training process and inference cycle. This "statelessness" arises from the model's reliance on batches of data as opposed to individual real-time data points, as demonstrated in Fig. 8.

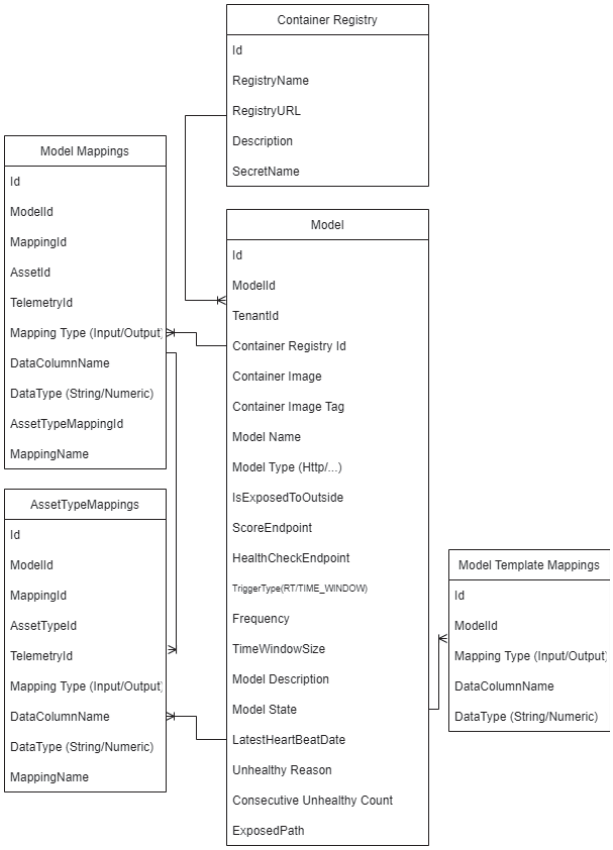

Fig. 7 Database schema for storing model information in SQL upon submission of a manifest file.

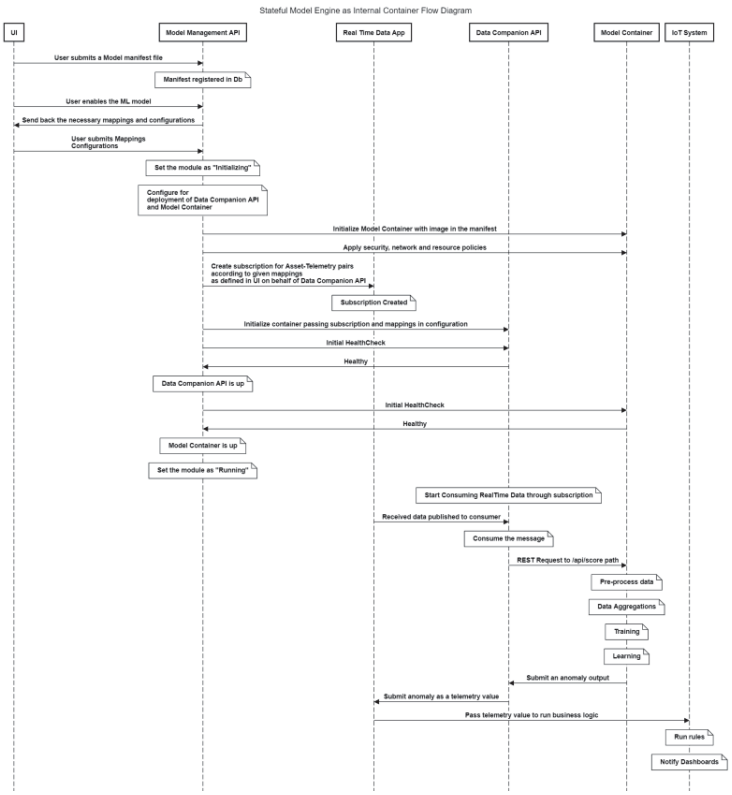

Fig. 8 Sequence diagram illustrating a stateful model completely hosted within the IoT system.

The stateless model's primary function lies in its ability to detect anomalies within a given dataset. It provides a level of control and predictability, making it a preferred choice in situations where absolute trust in the model's continual learning capabilities may be lacking.

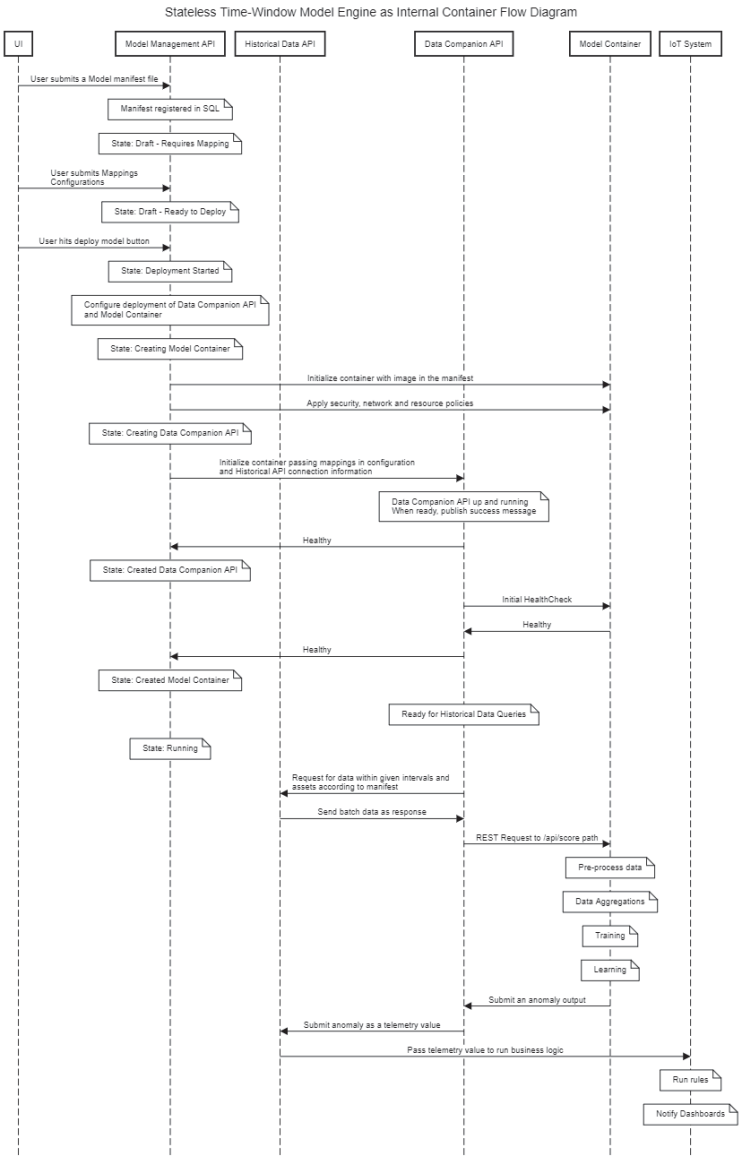

Fig. 9 Sequence diagram illustrating a stateless model completely hosted within the IoT system, providing full visibility of the state diagram.

Fig. 10 provides an illustrative depiction of how we can accommodate a model external to our system for various compelling reasons. These reasons may include scenarios where a model cannot be containerized or is too resource-intensive to be hosted within our IoT system. In such cases, all data-related operations are executed within this external model. However, a model container is designed to facilitate communication with our IoT system, ensuring the collection of data in compliance with established standards, and forwarding this data to the external model.

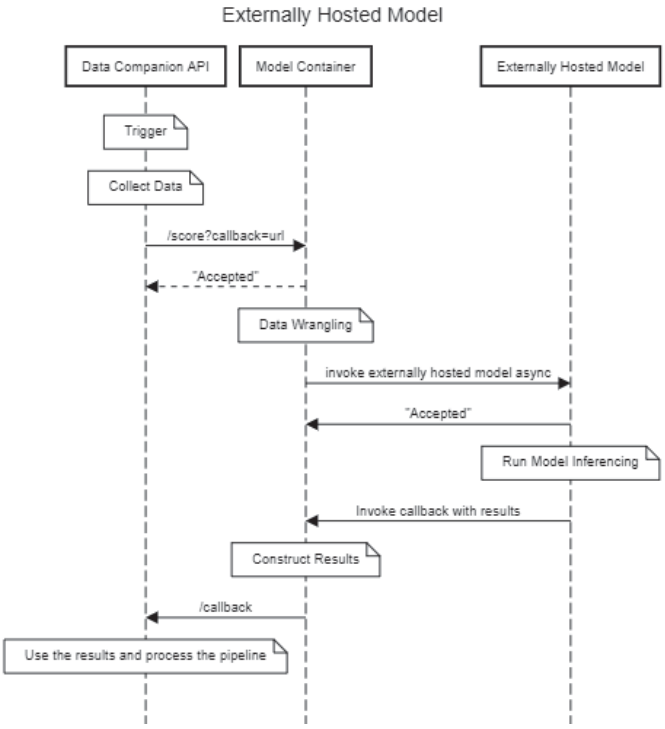

Fig. 10 Sequence diagram illustrating model logic hosted externally with Model Container acting as a data proxy for transmitting data to the external model, retrieving results, and integrating them into the system.

Within this diagram, another noteworthy feature is the use of callback URLs, which come into play when an asynchronous model is required. It is important to recognize that not all models can provide immediate data inference. Some models may necessitate significant time, potentially hours, to process extensive datasets or reach a suitable training threshold before they can commence inference operations. To address this, callback URLs are incorporated into the process. They are included in the requests both from the Data Companion API to the Model Container and from the Model Container to the Externally Hosted Model. This enables an asynchronous waiting mechanism, allowing other operations to proceed while awaiting the completion of the task. This approach ensures efficient handling of tasks that require extended processing times, contributing to a responsive and effective system.

This essentially encapsulates the fundamental workings of model management. The only requirement is that developers adhere to the specified object structure proposed by the Data Companion API when developing an endpoint and return data in the expected structure. Moreover, this proposal suggests the storage of simple yet generic models, such as temperature anomaly detection, in a separate repository along with their manifest files and corresponding model images. This facilitates the accessibility of these models to customers who may not have their own models yet. Customers can easily navigate the model library, select a basic model, and import its manifest file to initiate a new model for assets equipped with temperature sensors. While these pre-defined models may not achieve high precision, they provide quick and reusable solutions, thereby adding value in an efficient manner.

IV. Conclusion

In this study, we have introduced a resilient and adaptable framework for augmenting AIoT capabilities within an IoT product. This framework shifts the responsibility of model development to individual stakeholders, providing them with the flexibility to design and deploy models that suit their specific needs. Additionally, we offer a repository for fundamental yet shareable models contributed by tenants, fostering a collaborative environment that benefits all customers. For customers equipped with data science teams, the framework offers full autonomy to design and seamlessly integrate their custom models into our ecosystem. The only requirement is adherence to our established IoT system data provisioning standards, simplifying the integration process. As IoT product owners, we eliminate the necessity of building extensive in-house data science teams. Instead, we can establish strategic partnerships with external data science companies, allowing us to readily access reusable, generic, and highly precise models when required. This collaborative approach provides a valuable "nice-to-have" asset in our ongoing pursuit of enhancing our IoT product and delivering cutting-edge solutions to our customers.

Acknowledgment

This work is supported by Priva BVA.

References

[1] Dong, B., Shi, Q., Yang, Y., Wen, F., Zhang, Z., & Lee, C. (2021). Technology evolution from self-powered sensors to AIoT enabled smart homes. Nano Energy, 79, 105414.

[2] Nahr, J. G., Nozari, H., & Sadeghi, M. E. (2021). Green supply chain based on artificial intelligence of things (AIoT). International Journal of Innovation in Management, Economics and Social Sciences, 1(2), 56-63.

[3] Zonta, T., Da Costa, C. A., da Rosa Righi, R., de Lima, M. J., da Trindade, E. S., & Li, G. P. (2020). Predictive maintenance in the Industry 4.0: A systematic literature review. Computers & Industrial Engineering, 150, 106889..

[4] Sipola, T., Alatalo, J., Kokkonen, T., & Rantonen, M. (2022, April). Artificial intelligence in the IoT era: A review of edge AI hardware and software. In 2022 31st Conference of Open Innovations Association (FRUCT) (pp. 320-331). IEEE.

[5] Sun, Z., Zhu, M., Zhang, Z., Chen, Z., Shi, Q., Shan, X., ... & Lee, C. (2021). Artificial Intelligence of Things (AIoT) enabled virtual shop applications using self-powered sensor enhanced soft robotic manipulator. Advanced Science, 8(14), 2100230

[6] Chen, C. J., Huang, Y. Y., Li, Y. S., Chang, C. Y., & Huang, Y. M. (2020). An AIoT based smart agricultural system for pests detection. IEEE Access, 8, 180750-180761.

[7] Zhang, P., Chen, R. P., Dai, T., Wang, Z. T., & Wu, K. (2021). An AIoT-based system for real-time monitoring of tunnel construction. Tunnelling and Underground Space Technology, 109, 103766.

[8] Chen, S. W., Gu, X. W., Wang, J. J., & Zhu, H. S. (2021). AIoT used for COVID-19 pandemic prevention and control. Contrast media & molecular imaging, 2021.

[9] Mamza, E. S. (2021). Use of AIOT in health system. International Journal of Sustainable Development in Computing Science, 3(4), 21-30.

[10] De Prado, M., Su, J., Saeed, R., Keller, L., Vallez, N., Anderson, A., ... & Pazos Escudero, N. (2020). Bonseyes AI pipeline: bringing AI to you. ACM Transactions on Internet of Things.

[11] Barnes, J. (2015). Azure machine learning. Microsoft Azure Essentials. 1st ed, Microsoft.

[12] Das, A., Patterson, S., & Wittie, M. (2018, December). Edgebench: Benchmarking edge computing platforms. In 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion) (pp. 175-180). IEEE.

[13] S. Yao, Y. Hao, Y. Zhao, A. Piao, H. Shao, D. Liu, S. Liu, S. Hu, D. Weerakoon, K. Jayarajah, A. Misra, and T. Abdelzaher. 2019. Eugene: Towards Deep Intelligence as a Service. In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). 1630–1640

[14] Luksa, M. (2017). Kubernetes in action. Simon and Schuster.

[15] Rhea, S., Wang, E., Wong, E., Atkins, E., & Storer, N. (2017, May). Littletable: A time-series database and its uses. In Proceedings of the 2017 ACM International Conference on Management of Data (pp. 125-138).

[16] Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. International Journal of Computer Science and Network Security (IJCSNS), 17(3), 228.

[17] Python, W. (2021). Python. Python Releases for Windows, 24.

[18] Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. International Journal of Computer Science and Network Security (IJCSNS), 17(3), 228.

[19] Ferreira, A. P., & Sinnott, R. (2019, December). A performance evaluation of containers running on managed kubernetes services. In 2019 IEEE international conference on cloud computing technology and science (CloudCom) (pp. 199-208). IEEE.

[20] Aytaç, K., & Korçak, Ö. (2022, November). Multi-tenant management in secured iot based solutions. In 2022 32nd Conference of Open Innovations Association (FRUCT) (pp. 56-64). IEEE.

[21] Rudrabhatla, C. K. (2018). Comparison of event choreography and orchestration techniques in microservice architecture. International Journal of Advanced Computer Science and Applications, 9(8).