scientific reports

OPEN

Exploring dynamic RESTful API implementation in IoT environments using Docker

Ebenhezer Mabotha[1], Nkateko E. Mabunda[1], Ahmed Ali[1] & Baseem Khan[1,2,3]✉

Effective deployment solutions are essential for maximizing the capabilities of Internet of Things (IoT) devices and platforms. This study proposes a technique for enhancing the management, monitoring, and deployment of Internet of Things (IoT) devices, focusing on Dynamic RESTful APIs and Docker technologies. The suggested framework emphasizes reliable interaction and real-time flexibility between IoT devices and deployment infrastructures via a Dynamic RESTful API, combined with the deployment convenience given by Docker's lightweight containerization. The framework's applicability in real-world contexts was tested using an ESP8266 NodeMCU microcontroller and Raspberry Pi, both coupled with DHT11 sensor used to measure temperature and humidity readings. The devices' own ability to interact via built-in Wi-Fi, which enables data transfer and storage via HTTP requests, demonstrates the framework's usefulness in managing and deploying IoT devices. Furthermore, the API's dynamic nature, which allows for endpoint updates without requiring software modifications, provides an important feature for adaptive device behavior, solving key difficulties in IoT deployment, such as scalability and environmental condition changes. The results highlight the dynamic API's broad application and adaptability across various IoT devices, demonstrating its flexibility. This work adds to the body of knowledge on effective IoT deployment techniques while also laying the groundwork for future industry developments by establishing a framework for managing and deploying IoT devices.

Keywords RESTful, API, IoT, Docker, Deployment, Interface

Representational State Transfer (REST) has emerged as the dominant architectural strategy for developing interconnected applications, particularly APIs, due to its simplicity and ability to use current web standards and protocols[1]. RESTful APIs have been widely adopted because of their stateless form, which simplifies client–server interaction and enhances application scalability by allowing them to handle large numbers of concurrent queries. However, as the number of IoT devices increases, the requirement for efficient, adaptable, and flexible communication APIs becomes increasingly essential[2]. The traditional RESTful APIs, which commonly assume static endpoints and regular request patterns, frequently fall short in dynamic and extremely variable environments, such as those given by IoT devices.

This study provides a Dynamic RESTful API framework that is flexible and scalable in IoT deployments, which has been missing in the industry. The term dynamic deployments refers to the ability to create, update, and delete API endpoints and their corresponding database schemas at runtime without modifying the core application code. Instead of injecting code into IoT devices, our solution dynamically creates RESTful endpoints and database tables to handle new types of devices, sensors, or data structures. IoT systems require agile interfaces that can easily adapt to new functionality as devices and requirements evolve.

To simplify deployment, this study uses Docker, a modern containerization technology in order to streamline deployment. By isolating the API framework and its dependencies behind containers, Docker's lightweight, portable environments guarantee consistent deployment under a range of conditions. This encapsulation is perfect for managing different infrastructure configurations because it guarantees compatibility and streamlines deployment. Efficiency, portability, and application isolation are some of Docker's benefits, which result in more streamlined and dependable software development and deployment processes[3].

The study proposes a Dynamic RESTful API framework's architecture, design, and implementation details, including deployment strategies and endpoint and schema generation that make dynamic RESTful API provisioning and administration easier. Improved scalability, reduced integration problems, and increased interoperability are the benefits of this strategy, which makes meeting the demands of contemporary IoT

[1]Department of Electrical and Electronic Technology, University of Johannesburg, Johannesburg, South Africa. [2]Department of Electrical and Computer Engineering, Hawassa University, Hawassa, Ethiopia. [3]Center for Research on Microgrids (CROM), Huanjiang Laboratory, Zhuji, China. ✉email: baseem.khan04@gmail.com

deployments essential for effectively managing intricate and dynamic systems. The aforementioned method effectively tackles the three main issues of security, flexibility, and standardization by leveraging Docker.

This paper is divided into following number of sections. In Sect. "Related works", related work is examined, providing a thorough summary of earlier research and advancements that serve as the basis for this investigation. The methodology is described in Sect. "Methodology", along with the components, design, and techniques of the suggested framework. The results are presented and analyzed in Sect. "Experimental results", with an interpretation based on the study questions. The article is concluded in Sect. "Conclusion" with a summary of the main findings, thoughts on their significance, and suggestions for other research avenues.

Related works

The effectiveness and dependability of dynamic API approaches are significantly impacted by the difficulties they offer. One of the biggest problems is the absence of uniformity across various dynamic API implementations, which causes inconsistency and complexity in their design and use[1]. Developers find it difficult to accept and effortlessly incorporate dynamic APIs into their systems due to absence of standardization. Scalability issues come up, particularly when managing a large volume of requests or dynamic changes in system load, which can strain resources and reduce efficiency[4]. Security is still a major problem, and if dynamic APIs are not provided and guarded correctly, there is a serious danger of vulnerabilities like injection attacks or unauthorized access[5].

The use of RESTful APIs in the Internet of Things was reviewed by[6]. A thorough analysis of the use of RESTful APIs in the Internet of Things (IoT) space is presented by the author. The authors reviewed the literature in detail and investigated the several ways that RESTful APIs are used to help IoT devices and systems communicate and interact. Their flexibility, scalability, and interoperability with web technologies are highlighted as they go over the benefits and drawbacks of utilizing RESTful APIs in Internet of Things applications. The analysis also explores particular use cases and applications where RESTful APIs are crucial for facilitating smooth interoperability and integration inside IoT networks. Because JSON objects are lightweight and simple to understand, as demonstrated by the work of[7], the author further explains why the JSON format is typically chosen over XML in requests and responses.

The concepts and best practices necessary for creating interfaces that developers find appealing are described by the author in the work[8] on web API design. The author highlights how important it is to design APIs that are simple to use and comprehend in order to promote broad integration and adoption. He goes into detail on important ideas including using consistent and understandable URLs, choosing suitable HTTP methods, putting in place efficient error handling, and abiding by RESTful principles. The author intends to ensure that APIs meet technical requirements while also significantly enhancing the developer experience, which eventually aids in the success and maintainability of software projects, by offering practical advice. In keeping with the broader discussions on API design, this study builds on the work done by[9] and emphasizes developer happiness and usability as essential elements in the creation of successful online services.

The use of Python and FastAPI to build dynamic RESTful APIs is growing because of their dependability, effectiveness, and user-friendliness. As demonstrated by the work of[10], Python's vast ecosystem, which is full of libraries and frameworks, offers strong tools for the quick development and integration of a wide range of functionality. Famous for its remarkable speed and automatic interactive API documentation produced with OpenAPI and JSON Schema, FastAPI is a contemporary, high-performance web framework for creating APIs with Python 3.6+[11]. The asynchronous features of FastAPI make use of Python's asyncio package, which enables effective management of several concurrent requests and enhances performance under high demand. The development, creation, and operation of the FastAPI application, including file structure, deployment, and error handling, are covered in[12]. The dependency injection mechanism of FastAPI encourages component modularity and reusability, which is essential for sustaining large-scale applications. FastAPI guarantees reliable, self-documenting APIs by conforming to the OpenAPI standard, which improves maintainability and facilitates client integration[1]. All things considered, Python and FastAPI are excellent resources for creating dynamic RESTful APIs because they are scalable, performant, and simple to maintain.

Deployment complexity is also a major barrier, especially when managing dependencies across several environments or integrating dynamic APIs with current systems, as[13] discusses. Since dynamic APIs must be updated or altered frequently to meet growing requirements, maintaining them presents additional difficulties that may result in versioning and compatibility problems[14]. Dynamic configuration and processing performance overhead may impact system performance and reaction times, requiring optimization. Since dynamic APIs should be able to accommodate interface changes without interfering with already-existing functionality, the ability to adjust to changes in interface structure is also crucial[15].

Several strategies are used to address these issues. These consist of API composition, middleware and interceptors, dynamic data filtering and transformation, standard dynamic routing, and API gateway and proxy solutions[4,13]. The application management and deployment problems can be solved by using the application development and deployment strategies described by[12]. These techniques aim to improve dynamic APIs' performance, scalability, security, deployment, and maintenance over a range of application. In order to guarantee the efficiency and dependability of dynamic API implementations, thorough documentation and testing protocols are also necessary. Coordinated efforts in standardization, security, scalability, deployment techniques, maintenance plans, performance optimization, and documentation are required to guarantee the dependability and efficacy of dynamic API implementations in a range of application scenarios.

There are a number of dynamic API management solutions available, however each has drawbacks in contrast to our framework:

- *API gateways* Gateways like Kong or AWS API Gateway offer authentication, rate limiting, and load balancing[16,17]. However, they introduce latency from additional processing layers and require complex configurations for dynamic endpoints, unlike our lightweight FastAPI-based approach, which achieves low latency.
- *GraphQL* GraphQL's flexible querying reduces over-fetching in IoT systems[18–20], but its complex schema definitions and resolver overhead are impractical for resource-constrained devices. Our framework's RESTful, schema-less design ensures simplicity and efficiency.
- *gRPC* Built on HTTP/2, gRPC offers low-latency communication for IoT[21], but its reliance on protocol buffers limits compatibility with web-based IoT ecosystems using HTTP/1.1 and JSON[22]. Our RESTful framework ensures broader compatibility and ease of integration[23]. Niswar et al.[24] note gRPC's faster response times but higher CPU usage compared to REST, aligning with our focus on resource efficiency.
- *Traditional REST frameworks* Frameworks like Django, FastAPI, and Flask support dynamic routing at the URL level[25–27], but their static data models and ORM reliance limit schema adaptability. Manual updates increase operational overhead, whereas our schema-less, database-driven endpoint generation eliminates this need.

This study advances the state-of-the-art by introducing a framework that dynamically generates RESTful API endpoints and database schemas at runtime, suitable for IoT environments. Unlike prior work, which often assumes static endpoints or manual configuration, this approach automates endpoint creation and integrates it with Docker containerization for smooth deployment. This addresses critical gaps in scalability, adaptability, and ease of integration, making it particularly suited for dynamic IoT ecosystems.

Methodology

This section describes how to create and deploy a dynamic RESTful API framework. It focuses on the framework architecture, RESTful API architecture, data storage, deployment, and usage. The process of development follows best practices to attain scalability, flexibility, and maintainability for the framework, which is critical in ensuring efficient deployment and utilization of IoT systems. Figure 1 illustrates the framework architecture.

Framework design

The proposed framework consists of two main containers: the webserver and the Dynamic RESTful API container, which houses the core API application and database. The development of the framework and configuration of the containers are detailed in the following sections.

Docker containers

Microservices and containerization are the cornerstones of the design. As described in[28], Docker containers offer a lightweight and portable method for bundling application components, consistency and repeatability across

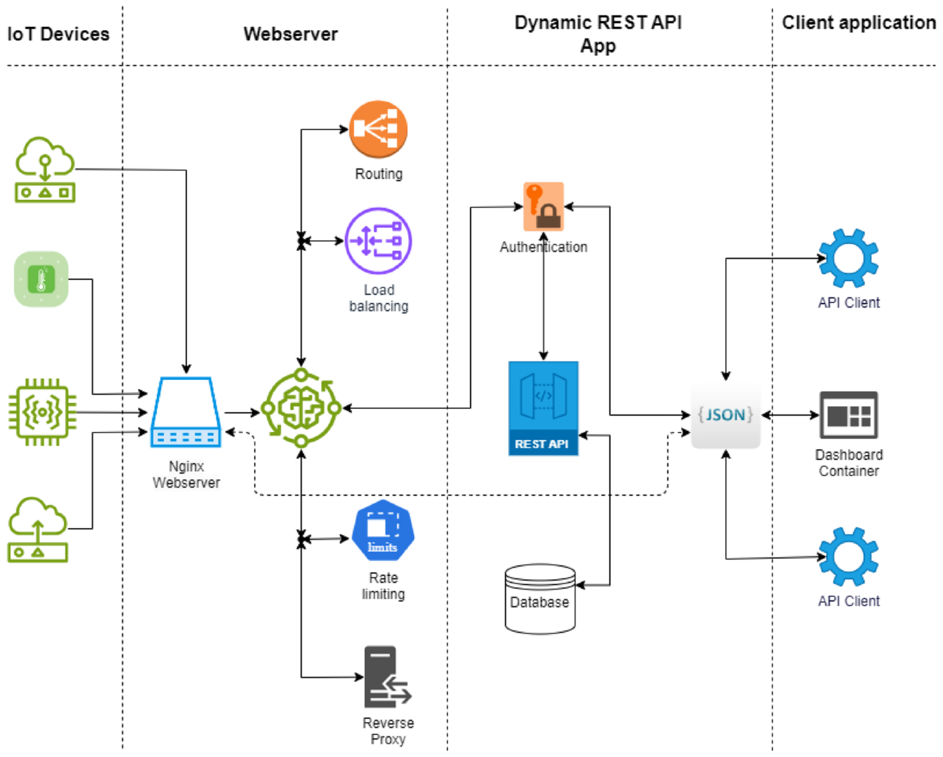

Fig. 1. Architectural design.

environments. In adaptive IoT contexts, this kind of containerization is essential to obtaining the necessary flexibility and efficiency. The following primary container components are used by the framework:

Webserver container
Nginx, which serves as the gateway and manages routing and rate limitation, is a component of the web server instance that the architecture depends on. This configuration makes use of a pre-built container that is based on the Docker Hub Nginx image, but it has been customized to enable rate limitations and particular routing for this deployment. As a reverse proxy, Nginx routes inbound requests from external users and Internet of Things devices to the appropriate backend services. Rate-limiting restrictions are applied to prevent abuse and improve system performance. Nginx is utilized in this framework since a study in[29] demonstrates that it performs better than Apache and other modern web servers.

Dynamic RESTful API (interface) container
In a containerized environment, the Python FastAPI application and PostgreSQL database are deployed using a Docker image created in accordance with standard procedures. Using Ubuntu 22.04's compatibility and stability, it was chosen as the basis image in accordance with reference[30]'s recommendations. Non-interactive package installations are carried out to reduce interruptions and expedite the build process. Installing iproute 2 and other essential components guarantees efficient container management. Setting up environment variables for PostgreSQL guarantees safe database management inside the container. The application files are copied and the /app directory is formed in accordance with containerized application best practices. Standard containerization techniques are used to adapt SQL scripts, expose ports, and install Python dependencies. In order to ensure effective container management and deployment procedures, PostgreSQL is finally activated, and the FastAPI application is started in compliance with container deployment protocols. By incorporating industry best practices and insights from[31] the framework architecture ensures resilience and dependability in the deployment and operation of IoT systems while permitting flexibility, adaptability, and scalability.

RESTful API application

Framework and language choice
Python was chosen to create the RESTful API because of its robust performance, broad library and framework ecosystem, and track record of dependability. It is especially well-suited for API development because to its versatility and simple connection with external systems. FastAPI was selected as the framework since it is a cutting-edge, high-performance choice made especially for creating APIs using Python 3.6 and higher. The automatic creation of interactive API documentation with OpenAPI and JSON Schema is one of its most notable advantages; it streamlines the development and debugging processes. As said in[11], by giving developers an easier method to explore and comprehend various API endpoints, such auto-generated documentation increases transparency and promotes clearer communication between clients and servers.

Support for Python's asyncio library by FastAPI also makes it able to serve several concurrent requests with effectiveness, improving performance under load, as indicated by[12]. This asynchronous ability is especially important in systems such as IoT, where systems respond to concurrent data exchanges from several connected systems.

RESTful best practices
A consistent and effective method of client–server communication is achieved by the Restful API application's strict adherence to RESTful best practices. In addition to authentication for Sign Up and Login, it facilitates Create, Read, Update, and Delete (CRUD) actions using six APIs. To guarantee that the API functions dependably and suitably in a range of situations, each endpoint is made to provide succinct answers while adhering to the HTTP status codes listed in the official HTTP specification[32]. The architecture of the API is built on strategies providing high reliability, flexibility, and maintainability, with special attention to file organization management, deployment strategy, and error handling. While effectively processing multiple requests simultaneously, this framework uses FastAPI's features to enhance the adaptability of the application to dynamic conditions. This design approach is critical for maintaining the reliability and efficiency of the API as it scales to handle more users and devices[33,34].

Dynamic endpoint implementation
Runtime creation, updating, and deletion of API endpoints without changing the main application code is made possible by the dynamic RESTful API framework. New database tables and API endpoints are created via this procedure, known as dynamic deployments, using user-defined JSON payloads that include endpoint URLs, descriptions, table names, and field schemas (such as field names, types, and constraints). For instance, a user can specify a new endpoint for a temperature sensor by sending a JSON payload, which causes a RESTful endpoint and matching database table to be created.

This implementation does not entail giving IoT devices executable code, such as Python scripts or CPU/mem commands. IoT devices use HTTP requests to provide data to either specified or dynamically constructed endpoints. The incoming data is stored in the PostgreSQL database, validated against the defined schema, and made available through CRUD operations by the FastAPI application. IoT devices will simply need to implement HTTP client capability thanks to this method, which is lightweight and widely supported (for example, using libraries like HTTPClient for ESP8266 or requests in Python for Raspberry Pi).

Error management and user experience

In order to guarantee that clients receive responses that are both clear and helpful, effective error management is important for the application. By providing clients immediate feedback on whether endpoint creation and other actions were successful or unsuccessful, this improves the user experience. By employing efficient methods, the program minimizes latency and continues to function reliably even while handling unforeseen problems[33].

Flexibility and maintainability

The proposed framework provides a flexible and manageable method for administering dynamic APIs. The API framework combines the asynchronous characteristics of FastAPI with Python's vast ecosystem to address typical issues like flexibility, frequent endpoint updates, and performance optimization. As the API develops to accommodate new requests and add new capabilities, this procedure guarantees that it will remain robust, effective, and easy to maintain. The process of developing a dynamic interface is shown in Fig. 2, which also shows how the application creates new endpoints on the fly and incorporates them into the current system. This dynamic capability is a key component of the proposed architecture, offering the flexibility required to support a diverse range of IoT applications and dynamic situations.

The dynamic endpoint creation process can be summarized as follows:

- *User request* A user sends a POST request to the /dynamic endpoint with a JSON payload specifying the endpoint details (e.g., URL, fields).
- *Schema validation* The FastAPI application validates the payload against predefined rules (e.g., valid field types, unique table names).
- *Table creation* A new PostgreSQL table is created with columns corresponding to the specified fields, plus an auto-generated id and epoch_time for tracking.
- *Endpoint registration* The new endpoint is registered in the API, allowing subsequent CRUD operations (e.g., POST to /dynamic/ < URL > for data submission).
- *Storage and validation* The endpoint schema is stored in a separate metadata table for future validation during updates or data submissions.

This process ensures flexibility, as new endpoints can be added or modified without redeploying the application, and maintainability, as the schema storage enables consistent data validation.

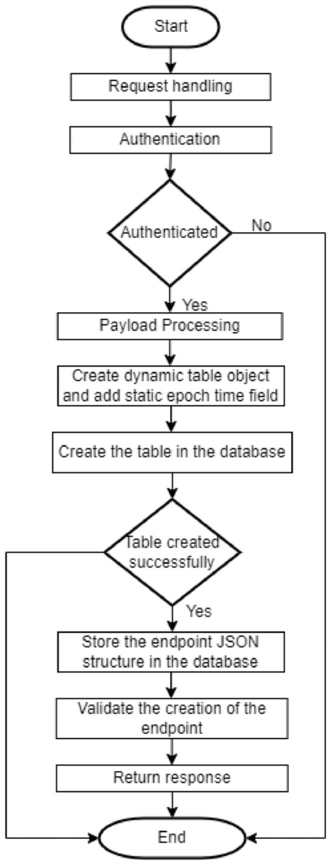
Start
Request handling
Authentication
Authenticated
No
Yes
Payload Processing
Create dynamic table object and add static epoch time field
Create the table in the database
Table created successfully
Yes
Store the endpoint JSON structure in the database
Validate the creation of the endpoint
Return response
End

Fig. 2. Add interface flow.

Database and storage

The selection of a database to use was motivated by factors including performance, scalability and ease of integration/usage with the technology stack. Based on the in-depth research provided in[35], PostgreSQL was the database management system of choice as it was very feature-rich, durable, and enjoyed excellent support in the Python environment. PostgreSQL support for JSONB data type was part of the reason it was selected for this framework. This feature allows flexible schema design and make it well suited for managing different and changing data structures that may be experienced in this context.

Data storage has been structured to align with the specific application needs which include support for CRUD operations on all endpoints and secure authentication workflow. The schema for the endpoints and user records have been designed to uphold data integrity and allow efficient, reliable data querying. The setup follows relational database principles and takes advantage of PostgreSQL's capabilities for handling complex data querying and multiple transactions. Atomicity, Consistency, Isolation, Durability (ACID) compliance, which ensures that data consistency and reliability during high-risk operations is one of the features that stood out as detailed by[36]. PostgreSQL's ease of integration with FastAPI via asynchronous database access to make it perform concurrent query handling strengthen the decision to use it in this context. This asynchronous approach adds to the framework's ability to scale, maintain stability and respond to queries even under demanding loads.

Deployment

Docker Compose was used in the deployment process to automate container initialization. Docker Compose facilitates the development and deployment of multi-container Docker applications, as illustrated in[37–39]. The docker compose file, which specifies the required services and configurations, is shown in summary form in Fig. 3.

The `docker-compose.yml` file in Fig. 3 defines two primary services:

Webserver container

- *Creates the Web server* The webserver container image is created using the Dockerfile found in the *./webserver* folder.
- *Exposes Ports* For HTTP traffic, it maps port 80 on the host to port 80 on the container; for HTTPS traffic, it maps port 443 on the host to port 443 on the container.

```
version: '3'
networks:
interface_network:
services:
  webserver:
    build: ./webserver
    ports:
      - "80:80"
      - "443:443"
    networks:
      - interface_network
  interface:
    build: ./interface
    ports:
      - "5432:5432"
      - "8888:8888"
    environment:
      - POSTGRES_PASSWORD=$POSTGRES_PASSWORD # Set your desired password here
    networks:
      - interface_network
```

Fig. 3. Docker-compose file.

- *Networks* Establishes a connection with the *interface_network*, allowing the webserver to communicate with other services.

Dynamic RESTful API container

- This service utilizes the Dockerfile found in the *./interface* folder to create the interface container service.
- Exposes ports: It maps host port 5432 to host port 5432 on the container for PostgreSQL database access and host port 8888 to container port 8888 for the FastAPI application.
- Environment variables: Sets environment variables, namely POSTGRES_PASSWORD, to configure the password for the PostgreSQL database.
- Networks: Establishes a connection with the *interface_network*, allowing the webserver to communicate with other service.

Data collection
Data was obtained using an ESP8266 NodeMCU microcontroller and Raspberry Pi 4. These devices were fitted with a humidity and temperature sensor. The Raspberry Pi was used to monitor the system metrics which include the Disk, CPU and Memory. These devices have Wi-Fi capabilities and enable them to make HTTP requests to the RESTful API to send the data. This setup enabled data collection and guaranteed that different data types and information from different IoT devices could be collected.

Performance evaluation
The process of evaluating the performance of the dynamic RESTful API followed a structured approach to ensure comprehensive testing of functionality, security, and scalability, tailored for IoT environments. Authentication testing verified secure access by simulating user registration and login processes. Upon login at the /login endpoint, a unique api_key is generated, required in request headers for subsequent API calls. Security was validated by testing unauthorized access attempts with invalid or expired api_keys, ensuring appropriate error responses and access denial.

Status code validation ensured endpoints returned correct HTTP status codes for various scenarios, including successful operations, unauthorized access, and validation failures. Test cases used valid and invalid inputs for POST, GET, PUT, and DELETE requests to verify behavior, such as data creation, retrieval, updates, and deletions, confirming adherence to REST standards for reliable client–server communication. Functional testing, conducted with Postman, validated endpoint operations (POST, GET, PUT, DELETE) for creating, retrieving, updating, and deleting dynamic interfaces. Responses were validated by comparing them to the expected status codes and payload content, ignoring Nginx rate limits to focus on functionality. Error-handling was tested with incorrect inputs to ensure robust exception management and relevant error messages.

Load testing was performed on the RESTful API framework to evaluate its performance when under high traffic volumes. Although the server was setup with a rate limiting rule (10 requests per second per IP), the setting was intentionally bypassed during testing to evaluate the API's true capacity beyond enforced constraints. The bypass permitted an initial high of 67.1 req/sec, which allowed stress testing to proceed above the specified threshold. Traffic was produced using JMeter at the intended rate of 600 requests per minute. During the 141-s test, which included continuous queries, metrics like error rates, throughput, and response times were recorded in a CSV file. This method gave important information on the scalability, resilience, and possible performance bottlenecks of the API in situations where typical operating boundaries are exceeded.

Experimental results
The following section demonstrates the outputs of the created system, highlighting the functionality of the framework, database functions and IoT data collection.

Dynamic RESTful API
The RESTful API application developed with FastAPI has demonstrated efficacy and functionality in dynamic environments. FastAPI is a very effective tool that provides automatic API documentation based on OpenAPI and JSON Schema. The API maximizes efficiency even during periods of high traffic by efficiently handling numerous requests at once by leveraging its asynchronous features. The API conforms to RESTful principles, providing CRUD operations through several endpoints, including Sign Up and Login authentication features. By efficiently responding and following HTTP status code guidelines, all endpoints guarantee seamless client–server communications. All of the endpoints for the dynamic API are listed in Table 1, and the sections that follow go into detail on the relevant functionality and methods for each endpoint.

Each endpoint's unique requests and responses are broken down in the tables below, which also offer details regarding the API's interaction methods:

Adding an interface
Adding a new interface/endpoint follows precise steps as shown in Fig. 2. Table 2 and Table 3 details the request and response information for this method.

Request See Table 2.

Response See Table 3.

Endpoint	Method	Description
/dynamic	POST	Register a new dynamic endpoint
/dynamic	GET	List all available dynamic endpoints
/dynamic	PUT	Edit settings or structure of an endpoint
/dynamic	DELETE	Remove an endpoint from the system
/dynamic/ < URL >	POST	Submit new data to an endpoint
/dynamic/interface/ < URL >	GET	Fetch stored data from an endpoint

Table 1. API endpoints for managing dynamic IoT interfaces.

Field	Value	Type
URL	'/dynamic'	
Method	POST	
Content-Type	application/json	
Payload		json
	interface_url	text
	interface_description	text
	table_name	text
	fields	list
	Inside 'fields' array	
	field_name	text
	field_type	text
	trendable	bool
	required	bool

Table 2. Request format for creating a dynamic endpoint.

Field	Value	Type
URL	'/dynamic'	
Status code	200	
Content-type	application/json	
Payload		json
	interface_id	integer
	interface_url	text

Table 3. Response after creating a dynamic endpoint.

Field	Value	Type
URL	'/dynamic'	
Method	GET	
Content-Type	application/json	
Payload		json
	interface_id	integer

Table 4. Request format for retrieving endpoints.

Retrieve interface
The **GET** method can be used to retrieve the user's dynamic interfaces. When the application receives a request, it retrieves every interface from the database for the authenticated user. If something goes wrong during this process, an error message is returned. The request and response data for this method are shown in Table 4 and Table 5.

Request See Table 4.

Response See Table 5.

Field	Value	Type
URL	'/dynamic'	
Status code	200	
Content-type	application/json	
Payload		json
	interface_url	text
	interface_id	integer
	interface_description	text
	table_name	text
	fields	list
	Inside 'fields' array	
	field_name	text
	field_type	text
	trendable	bool
	required	bool

Table 5. Response fields when retrieving endpoints.

Field	Value	Type
URL	'/dynamic'	
Method	PUT	
Content-type	application/json	
Payload		json
	interface_id	integer
	interface_description	text
	fields	list
	Inside 'fields' array	
	field_name	text
	field_type	text
	trendable	bool
	required	bool

Table 6. Request format for editing an endpoint.

Field	Value	Type
URL	'/dynamic'	
Status code	200	
Content-type	application/json	
Payload		json
	success	text

Table 7. Response after editing an endpoint.

Edit interface
Like the **POST** method, the **PUT** method of the "**/dynamic**" route adds new fields to the table or deletes the ones that aren't in the new body/payload, but it does not create a new endpoint. The modified payload will be saved in the database to be utilized for validation. An example of the edit request and response are shown in Tables 6 and 7.

Request See Table 6.

Response See Table 7.

Delete interface
The specific interface will be deleted using the **DELETE** function of the "**/dynamic**" route. After receiving a request, the endpoint authenticates the user using an API key. If authentication fails, an appropriate error message is returned. The application then retrieves the endpoint's identity from the request payload to verify that

Field	Value	Type
URL	'/dynamic'	
Method	DELETE	
Content-Type	application/json	
Payload		json
	interface_id	integer

Table 8. Request format for deleting an endpoint.

Field	Value	Type
URL	'/dynamic'	
Status Code	200	
Content-Type	application/json	
Payload		json
	success	text

Table 9. Response after deleting an endpoint.

Field	Value	Type
URL	'/dynamic/ < URL > '	
Method	POST	
Content-Type	application/json	
Payload		json
	< dynamic_param_1 >	integer
	< dynamic_param_2 >	text
	< dynamic_param_3 >	bool
	…	…

Table 10. Request format for adding data to an endpoint.

Field	Value	Type
URL	'/dynamic'	
Status Code	200	
Content-Type	application/json	
Payload		json
	success	text

Table 11. Response after adding data to an endpoint.

the interface is present. The matching interface data is then retrieved by accessing the database. The related table is then taken out of the database and the interface entry is removed. Details of the delete procedure are shown in Table 8 and Table 9.

Request See Table 8.

Response See Table 9.

Add endpoint data
The history or device records can be added using this method. Any additional information or entries can be sent using the URL string supplied when the endpoint was created. The "**/dynamic/ < URL >** " route can be reached by submitting a **POST** request. The application confirms the user's information before returning a response. Details of the request and response are shown in Tables 10 and 11.

Request See Table 10.

Response See Table 11.

Field	Value	Type
URL	'/dynamic/interface/ < URL > '	
Method	GET	
Content-Type	application/json	
Payload	None	

Table 12. Request format for retrieving endpoint history.

Field	Value	Type
URL	'/dynamic//interface/ < URL > '	
Status code	200	
Content-type	application/json	
Payload		json
	< dynamic_param_1 >	integer
	< dynamic_param_2 >	text
	< dynamic_param_3 >	bool
	…	…

Table 13. Response structure for endpoint history.

Endpoint	Method	# Samples	Success rate (%)
/login	POST	908	100
/dynamic	POST	100	100
/dynamic	GET	3872	100
/dynamic	PUT	2605	100
/dynamic	DELETE	100	100
/dynamic/interface/ < URL >	POST	3212	100
/dynamic/interface/ < URL >	GET	923	100

Table 14. Functional test results for all API endpoints.

Get endpoint history

A **GET** request to the "**/dynamic/interface/**" route will retrieve the interface information and history. The application checks the user's information before returning a response. The information about the request and response is shown in Tables 12 and 13.

Request See Table 12.

Response See Table 13.

Performance evaluation results

The framework's ability to dynamically generate and manage API endpoints at runtime addresses the need for adaptable deployments in IoT systems, where device types and data structures evolve rapidly. Unlike static RESTful APIs, our approach eliminates manual code updates, enabling seamless integration of new devices or sensors, as validated by the performance metrics below. These results highlight the framework's novelty in achieving low latency, high reliability, and minimal resource usage, critical for resource-constrained IoT ecosystems.

The framework was tested for functionality across dynamic endpoints supporting CRUD operations and authentication (Sign Up, Login). Functional testing ensured consistent responses and HTTP status codes, validating reliability for IoT applications. The results, shown in Table 14, confirm a 100% success rate across all endpoints, demonstrating robustness and correctness.

The /login endpoint averaged 127 ms across 908 samples, reflecting reliable but optimizable authentication processing. The /dynamic endpoints achieved exceptional efficiency, with GET requests averaging 19 ms (3872 samples), POST and DELETE at 53 ms (100 samples each), and PUT at 36 ms (2605 samples). The /dynamic/interface/ < URL > endpoints averaged 33 ms for POST (3212 samples) and 119 ms for GET (923 samples), demonstrating fast data handling. These low response times, particularly for dynamic endpoints, highlight the framework's suitability for real-time IoT applications.

Response time testing measured average latency across endpoints, providing insights into performance under typical IoT workloads. Results are summarized in Table 15.

Endpoint	Method	# Samples	Average response time (ms)
/login	POST	908	127
/dynamic	POST	100	53
/dynamic	GET	3872	19
/dynamic	PUT	2605	36
/dynamic	DELETE	100	53
/dynamic/interface/ < URL >	POST	3212	33
/dynamic/interface/ < URL >	GET	923	119

Table 15. Average response times for API endpoints.

Metric	Value
Total requests	4732
Successful requests	4585
Failed requests	147 (3.11% error rate)
Average response time	88.69 ms
Max response time	760 ms
Min response time	5 ms
Throughput	9.83 requests/sec

Table 16. Load testing results of the RESTful API framework.

A single thread with API key authentication was used to generate 4732 requests over 141 s of load testing. To test the API's performance under heavy load, the Nginx rate limit of 10 requests/sec was circumvented. Table 16 displays the results.

Container implementation
The two primary containers utilized in the architecture are the Webserver and the Interface. The Webserver manages the requests, while the Interface serves as the main container that permits API interactions, as detailed in the methodology section. Regardless of the common API client, the Interface container provides efficiency and versatility. Below are the results of the containers made following the method:

Webserver container
The official Nginx image from Docker Hub was used to build the webserver container; however, the default configuration was changed to a customized version that includes rate limitations and routing tailored to this application. An example of the custom Nginx configuration is illustration in Fig. 4.

Interface container
To show the Interface container's potential in Internet of Things projects, the application was deployed and assessed. Reliability, security, and flexibility were emphasized in the results.

The following significant findings were reached after the framework was used to develop dynamic interfaces for Raspberry Pi and ESP8266 devices:

i User authentication and security

- *Authentication* Users were successfully authenticated using the Sign-up and Login endpoints.
- *Rate limiting* To guard against misuse and guarantee equitable use, the Nginx webserver implemented rate-limiting regulations.

ii Constructing the dynamic endpoints

- *Endpoint creation process* New dynamic interfaces were created using fields that were supplied, including "***field_name***" "***field_type***" "***trendable***" and "***mandatory***" for data validation. To facilitate future validation during modifications, the application saved endpoint details, created new database tables, and validated user credentials.
- *Error management* The dynamic endpoint creation feature's dependability and user experience were improved with explicit error notifications.

Framework application: Raspberry Pi and ESP8266 interfaces
In order to illustrate the usefulness of the Dynamic RESTful API architecture, two dynamic endpoints/interface were added for Raspberry Pi and ESP8266 devices.

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;
server {
  listen       80;
   listen  [::]:80;
   server_name  localhost;

  # Preserve client information in variables
  proxy_set_header Host $host;
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
  proxy_set_header User-Agent $http_user_agent;
  proxy_set_header Referer $http_referer;
  proxy_set_header Authorization $http_authorization;
  proxy_set_header Cookie $http_cookie;
  location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
   }
   location /interface/ {
         limit_req zone=mylimit;
         proxy_pass http://interface:8888/;
   }
  # added error page
  error_page 404 = @notfound;
  location @notfound {
         root   /usr/share/nginx/html;
         index  index.html;

         try_files $uri $uri/ /index.html?$args;
   }

  error_page   500 502 503 504  /50x.html;
  location = /50x.html {
         root   /usr/share/nginx/html;
  }
}
```

Fig. 4. Nginx configuration.

Raspberry Pi interface
This endpoint was created to monitor the Raspberry Pi's system metrics, including DISK, RAM and CPU usage. The endpoint will later be modified. Figure 5 illustrates the JSON payload used to create the dynamic endpoint.

ESP8266 interface
This endpoint was created to measure temperature and humidity data from ESP8266 sensor. Figure 5 illustrates the JSON payload used to create the dynamic endpoint.

Figure 6 illustrates the ESP8266 endpoint payload. The raw interface information is stored in a separate table. This information is used to verify the request responses and return appropriate errors when necessary. For example, if a user tries to update the interface or add a new record, the application will use this information to validate each parameter and its type. Table 17 displays the raw dynamic table information from the database.

Hardware setup
The DHT11 sensor's temperature and humidity readings were fed into the Esp8266 in order to evaluate the framework. Temperature readings were specifically recorded using pin D5 of the ESP8266 microcontroller, which was linked to ground and 3.3 V VCC. Following data collection, the HTTP protocol was used to send the data to the designated endpoint, "**/dynamic/esp**".

The Raspberry Pi was configured to run Python 3.6 and Ubuntu 22.04. A Python script was created to make collecting system information, temperature, and humidity data easier. The DHT11 sensor data was read by the script using pin 7 (GPIO 4), while pin 1 (3.3 V) supplied power and pin 6 supplied ground (GND). After that, the specified API endpoint received the gathered data. To make data retrieval easier, the cronjob scheduling system was used to call the Python script.

```json
{
    "interface_url": "/raspberrypi",
    "interface_description": "Endpoint for the Pi ",
    "table_name": "t_pi",
    "fields":[
        {
            "field_name": "cpu",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        },
        {
            "field_name": "disk",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        },
        {
            "field_name": "ram",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        }

    ]
}
```

Fig. 5. Raspberry Pi's endpoint payload.

```json
{
    "interface_url": "/esp",
    "interface_description": "Endpoint for the esp8266 weather station",
    "table_name": "t_esp",
    "fields":[
        {
            "field_name": "temperature",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        },
        {
            "field_name": "humidity",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        }

    ]
}
```

Fig. 6. ESP8266 endpoint payload.

id	interface_id	interface_url	interface_description	table_name	fields	interface_owner
4	1709138487598	/raspberrypi	Endpoint for the Pi	t_pi	{"{"field_name": "ram", "field_type": "Integer", "trendable": true, "required": true}","{"field_name": "cpu", "field_type": "Integer", "trendable": true, "required": true}","{"field_name": "disk", "field_type": "Integer", "trendable": true, "required": true}","{"field_name": "epoch_time", "field_type": "Integer", "trendable": false, "required": false}"}	2
39	1709830425232	/esp	Endpoint for the esp8266 weather station	t_esp	{"{"field_name": "temperature", "field_type": "Integer", "trendable": true, "required": true}","{"field_name": "humidity", "field_type": "Integer", "trendable": true, "required": false}","{"field_name": "epoch_time", "field_type": "Integer", "trendable": false, "required": false}"}	2

Table 17. Database metadata for dynamic endpoints.

id	ram	cpu	disk	epoch_time
38710	31	0	51	1716038767
38708	31	0	51	1716038526
38706	31	0	51	1716038286
38703	31	0	51	1716037926

Table 18. Raspberry Pi dynamically generated database table.

id	temperature	humidity	epoch_time
13332	24	32	1716039303
13331	24	32	1716039007
13330	24	33	1716038713
13329	24	32	1716038403

Table 19. Esp8266 database table.

Trend data

For both the Raspberry Pi and the ESP8266, JSON data was obtained via the dynamic RESTful API in order to examine the outcomes. The data was then visualized by creating plots with Matplotlib. Tables 18 and 19, respectively, show the database's dynamically generated tables for the Raspberry Pi and ESP8266. These tables include a selection of the information that is kept in the database.

Raspberry Pi data The dynamically generate table for the Raspberry Pi with sample data is shown in Table 18. The collected system metrics from the device are visualized in Fig. 7, Figs. 8 and 9. These representations provide clear insights into the information that were being monitored by the devices.

ESP8266 data The dynamically generate table for the ESP8266 with sample data is illustrated in Table 19. Figures 10 and 11 below show the temperature and humidity data from the ESP. Clear insights into the information being monitored by the device were provided by these visual representations.

The Raspberry Pi was fitted with a temperature and humidity sensor in order to evaluate the RESTful API's adaptability. The temperature and humidity parameters were added to the Raspberry Pi interface in order to achieve this. Figure 12 shows the JSON payload that was used to update the interface.

The features of the Raspberry Pi interface were extended with the incorporation of a temperature and humidity sensor. Temperature and humidity readings were added to the interface, allowing additional analysis of data and providing useful information. The ability to easily extend support for new devices and sensors or additional parameters highlights the API framework's versatility and adaptability. The major advantage is that such features can be added without changing the existing system architecture, demonstrating that the API is sufficiently robust and flexible to accommodate a variety of changing requirements. Figures 13 and 14 present the updated statistics for the Raspberry Pi, including the temperature, and Table 20 presents the updated database table and sample records.

Conclusion

This study's major goal was to create and assess a dynamic RESTful API framework for the flexible, scalable, and maintainable deployment and management of IoT systems. We employed a microservices architecture and Docker's containerization techniques to accomplish these goals. In order to provide seamless integration and expansion of system capabilities while preserving the fundamental application structure, the framework was designed to provide dynamic endpoints.

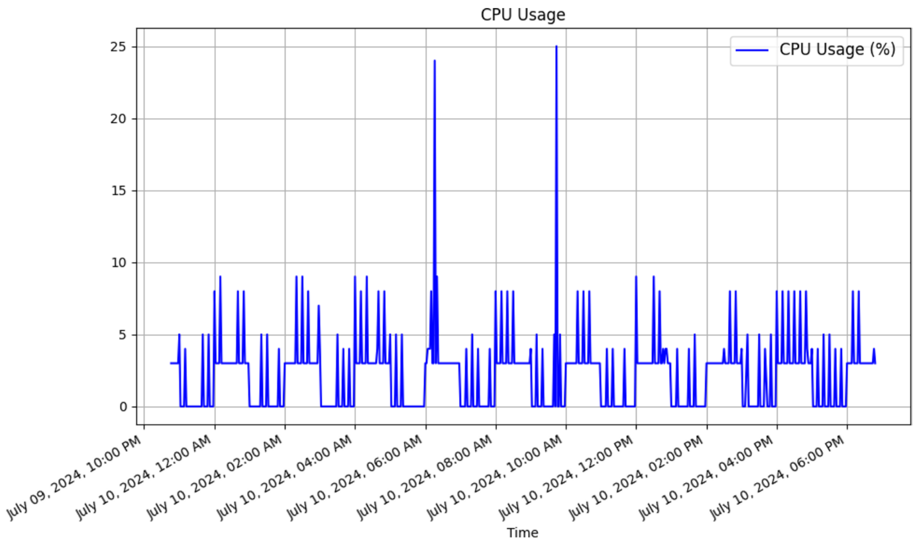

Fig. 7. Raspberry Pi CPU usage graph.

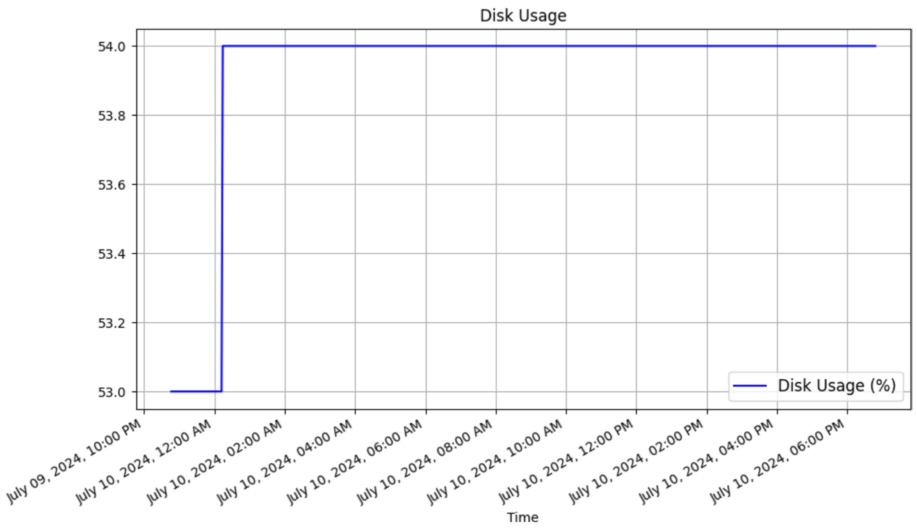

Fig. 8. Raspberry Pi DISK graph.

Python and FastAPI were selected for the RESTful API application because of its simplicity and broad library ecosystem. The asynchronous features of FastAPI and the Python asyncio module allowed for the simultaneous processing of many requests, which enhanced the API's performance under high load. The application extensively followed RESTful best practices to support CRUD operations across many endpoints and dynamic interface development. The ability to design new interfaces with specific parameters showed how flexible the framework was.

PostgreSQL was selected as the database management system because of its solid integration within the Python ecosystem and wide range of features, which ensured that the application would quickly satisfy its operational requirements. PostgreSQL's dedication to FastAPI enables asynchronous database access, improving scalability and performance while ensuring responsiveness and reliability for a range of applications.

To assess the framework's versatility, a temperature and humidity sensor was added to the Raspberry Pi interface. This improved both the Raspberry Pi measurements and the system matrices. This integration demonstrates how easy additional functionality may be implemented in the system. The update required minor changes to the Raspberry Pi's Python code but did not alter the framework's code. This smooth transition demonstrates the API's capacity to manage changing requirements without causing significant interruptions, achieving the study's main goals.

The performance evaluation results revealed that the API had an efficient average response time of 88.69 ms when handling 4732 requests over 141 s, with a throughput of 9.83 requests per second, despite an initial surge to 67.1 requests per second after bypassing the rate limit to assess its core performance. However, a 3.11% error rate and response time peaks of up to 760 ms showed potential server-side bottlenecks, such as resource

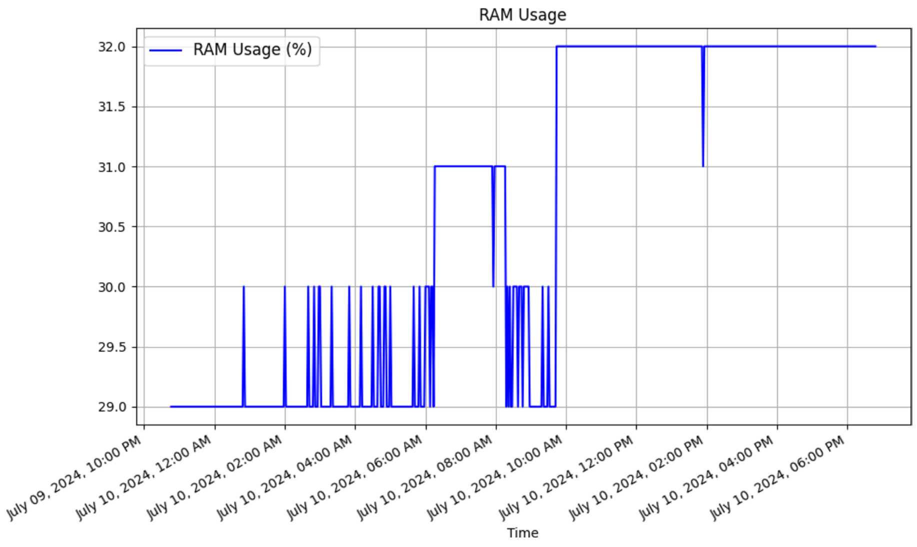

Fig. 9. Raspberry Pi RAM graph.

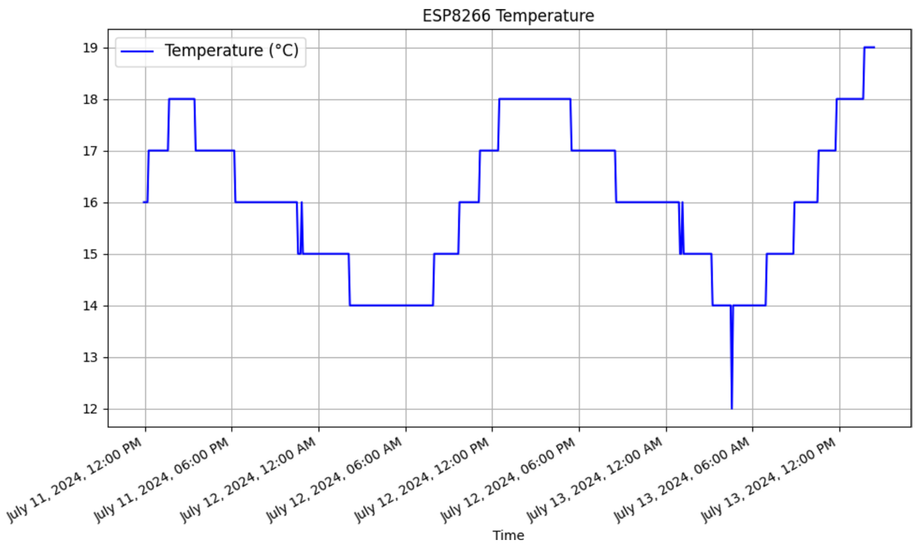

Fig. 10. ESP8266 temperature graph.

constraints, highlighting opportunities for improvement to increase reliability under high load. Response time testing further validated the API's efficiency, with endpoints maintaining a reasonable average response time across various samples under typical load conditions, affirming its effectiveness for user authentication tasks in IoT environments.

In this regard, the study succeeded in developing a dynamic RESTful API architecture that can be used in IoT systems, demonstrating flexibility, adaptability, and maintainability. These findings demonstrate the potential of the proposed framework for effectively managing and deploying IoT systems while fulfilling the dynamic and developing needs of present applications.

The future scope of this study includes expanding the framework by integrating a pub/sub protocol such as MQTT to fully adhere and accommodate other communication protocols. Additionally, the framework requires a method of adding and processing alerts and sending notifications. A container can be added to the framework to solely address this problem without interfering with the existing implementation. By incorporating these aspects, the framework can potentially evolve into a comprehensive and advanced solution for the deployment of dynamic infrastructure. This will also make it more useful for various applications and industries.

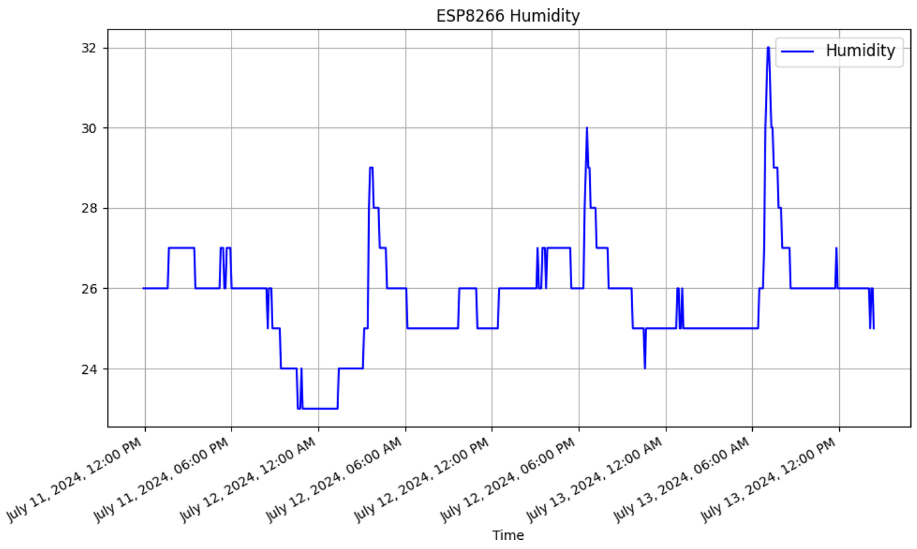

Fig. 11. ESP8266 humidity graph.

```json
{
    "interface_id": 1709138487598,
    "fields":[
        {
            "field_name": "ram",
            "field_type": "Integer",
            "trendable": true,
            "required": true
        },
        {
            "field_name": "cpu",
            "field_type": "Integer",
            "trendable": true,
            "required": true
        },
        {
            "field_name": "disk",
            "field_type": "Integer",
            "trendable": true,
            "required": true
        },
        {
            "field_name": "temperature",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        },
        {
            "field_name": "humidity",
            "field_type": "Integer",
            "trendable": "true",
            "required": "true"
        }
    ]
}
```

Fig. 12. Raspberry Pi update payload.

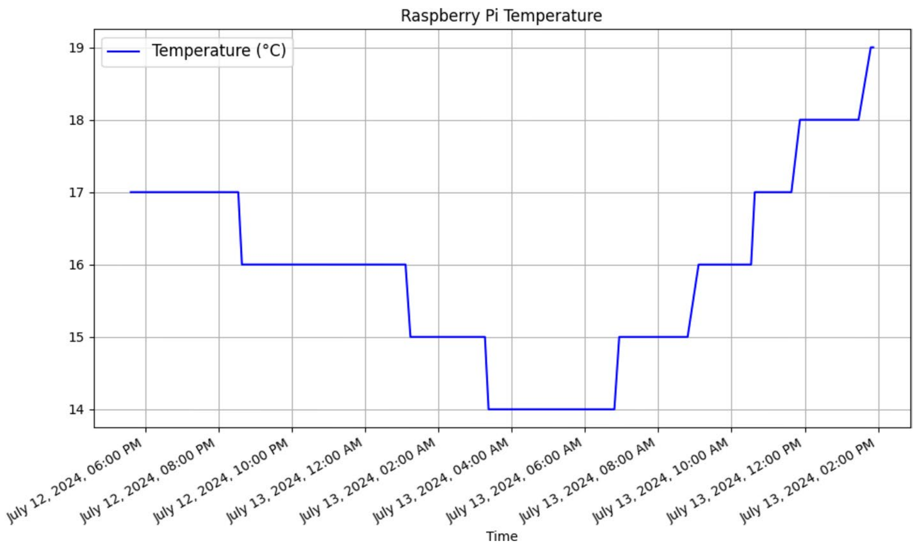

Fig. 13. Raspberry Pi temperature graph.

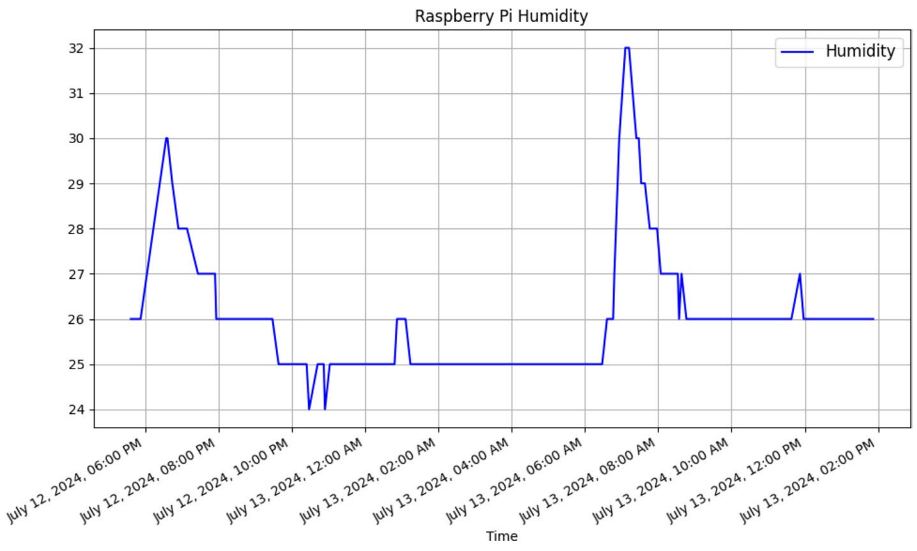

Fig. 14. Raspberry Pi humidity graph.

id	ram	cpu	Disk	epoch_time	temperature	humidity
38710	31	0	51	1716038767	24	33
38708	31	0	51	1716038526	24	33
38706	31	0	51	1716038286	24	33
38703	31	0	51	1716037926	24	33

Table 20. Raspberry Pi updated database table.

Data availability
The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Received: 10 August 2024; Accepted: 14 August 2025
Published online: 01 October 2025

References

1. Fielding, R. T. Architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine (2000).
2. Richardson, L. & Ruby, S. *RESTful Web Services* (O'Reilly Media Inc., 2007).
3. Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* **2014**(239), 2–11 (2014).
4. Mehta, B. *RESTful Java Patterns and Best Practices* (Packt Publishing, 2014).
5. Daigneau, R. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* (Addison-Wesley Professional, 2011).
6. Shaikh, N. et al. Application of RESTful APIs in IOT: A review. *Int. J. Res. Appl. Sci. Eng. Technol.* **9**, 1022214 (2021).
7. Crockford, D. (2006). JSON: The fat-free alternative to XML. in *Proceedings of XML 2006 (pp. 1–1). Boston, USA*. Retrieved from http://www.json.org/fatfree.html (2006).
8. Mulloy, B. *Web API design: Crafting interfaces that developers love* (2012).
9. Vinoski, S. RESTful web services development checklist. *IEEE Internet Comput.* **12**(6), 94–96 (2008).
10. Montenegro, H. & Gómez, J. *Building APIs with Python and FastAPI* (Apress, 2020).
11. Ramírez, S. *FastAPI documentation*. Retrieved from https://fastapi.tiangolo.com/ (2024).
12. Lathkar, M. *High-Performance Web Apps with FastAPI: The Asynchronous Web Framework Based on Modern Python* (Packt Publishing, 2021).
13. Richardson, L. & Amundsen, M. *RESTful Web APIs: Services for a Changing World* (O'Reilly Media, Inc., 2013).
14. Stopford, B. *Designing Event-Driven Systems* (O'Reilly Media, 2018).
15. Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). RESTful web services vs. "big" web services: Making the right architectural decision. in *Proceedings of the 17th International Conference on World Wide Web* (pp. 805–814) (2008).
16. Gadge, S., & Kotwani, V. *Microservice architecture: API gateway considerations*. GlobalLogic Organisations, 11 (2017).
17. Zhao, J. T., Jing, S. Y. & Jiang, L. Z. Management of API gateway based on micro-service architecture. *J. Phys. Conf. Ser.* **1087**(3), 032032 (2018).
18. Ed-Douibi, H., Izquierdo, J. L. C. & Cabot, J. *Automatic generation of test cases for REST APIs: A specification-based approach*. in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 181–190). https://doi.org/10.1109/EDOC.2018.00031 (2018).
19. Hartig, O., & Pérez, J. Semantics and complexity of GraphQL. in *Proceedings of the 2018 World Wide Web Conference (WWW '18)* (pp. 1155–1164). Lyon, France (2018).
20. Quiña-Mera, A., Fernandez, P., García, J. M. & Ruiz-Cortés, A. GraphQL: A systematic mapping study. *ACM Comput. Surv.* **55**(10), 1–35 (2023).
21. Pamadi, V. N., Chaurasia, A. K. & Singh, T. Comparative analysis of gRPC vs. ZeroMQ for fast communication. *Int. J. Emerg. Technol. Innov. Res.* **7**(2), 937–951 (2020).
22. Chamas, C. L., Cordeiro, D. & Eler, M. M. Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis. in *Proc. 2017 IEEE 9th Latin-American Conf. Commun. (LATINCOM)*, pp. 1–6 (2017).
23. Anderson, J. & Brown, S. Containerization in API testing: Leveraging Docker for efficient and scalable test environments. *IEEE Softw.* **37**(1), 24–31. https://doi.org/10.1109/MS.2019.2938127 (2020).
24. Niswar, M., Safruddin, R. A., Bustamin, A. & Aswad, I. Performance evaluation of microservices communication with REST, GraphQL, and gRPC. *Int. J. Electron. Telecommun.* **70**(2), 429–436 (2024).
25. Flask, "Welcome to Flask," Flask Documentation. [Online]. Available: https://flask.palletsprojects.com/
26. FastAPI, "FastAPI – The fast web framework for building APIs with Python 3.7+," [Online]. Available: https://fastapi.tiangolo.com/
27. Django, "Django Web Framework," [Online]. Available: https://www.djangoproject.com/
28. Felter, W., Ferreira, A., Rajamani, K. & Rubio, J. An updated performance comparison of virtual machines and Linux containers. in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 171–172) (2015).
29. Kunda, D., Chihana, S., & Sinyinda, M. Web server performance of apache and Nginx: A systematic.
30. Pahl, C. Containerization and the paas cloud. *IEEE Cloud Comput.* **2**(3), 24–31 (2015).
31. Newman, S. *Building Microservices: Designing Fine-Grained Systems* (O'Reilly Media, Inc., 2015).
32. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. Berners-Lee, T. RFC 2616: Hypertext Transfer Protocol—HTTP/1.1. Retrieved from http://tools.ietf.org/html/rfc2616 (1999).
33. Higginbotham, J. *The Principles of API Design* (O'Reilly Media, Inc., 2018).
34. Voron, F. *Building Data Science Applications with FastAPI: Develop, manage, and deploy efficient machine learning applications with Python* (Packt Publishing Ltd., 2023).
35. Makris, A., Tserpes, K., Spiliopoulos, G., Zissis, D. & Anagnostopoulos, D. MongoDB Vs PostgreSQL: A comparative study on performance aspects. *GeoInformatica* **25**, 243–268 (2021).
36. Stonebraker, M., Rowe, L. & Wong, E. The design of POSTGRES. in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 340–355 (2007).
37. Docker Documentation. (n.d.). Docker Compose Overview. Retrieved from https://docs.docker.com/compose/.
38. Gkatziouras, E. *A Developer's Essential Guide to Docker Compose: Simplify the Development and Orchestration of Multi-container Applications* (Packt Publishing Ltd., 2022).
39. Smith, R. *Docker Orchestration* (Packt Publishing Ltd., 2017).

Author contributions

EM, NEM, AA and BK wrote the main manuscript text. EM, NEM, AA and BK prepared figures. All authors reviewed the manuscript.

Declarations

Competing interests

The authors declare no competing interests.

GitHub Repository

The GitHub repository has been fully populated with source code, documentation, and supporting materials. It is accessible here: https://github.com/Ebenhezer/dynamic_rest/

Additional information

Correspondence and requests for materials should be addressed to B.K.