

A Scalable Service Architecture with Request Queuing for Resource-Intensive Tasks

Kasidis Chaowvasin*, Pun Sutanchaiyanonta[†], Nont Kanungsukkasem[‡], Teerapong Leelanupab[§]

Faculty of Information Technology

King Mongkut's Institute of Technology Ladkrabang (KMITL), Bangkok, Thailand. 10520

Email: *kasidis.ch@outlook.com, [†]pun.sutan@gmail.com, [‡]nont@it.kmitl.ac.th, [§]teerapong@it.kmitl.ac.th

Abstract—Deploying Machine Learning (ML) prediction or Data Analytic (DA) process as a service in a Web API is not a trivial task. A number of settings and dependency requirements must be met to provide ML or DA successful solutions. In addition, an application that utilizes such an API needs to be always available to serve multiple users who can concurrently submit their requests. ML modeling or DA processing is a resource-intensive task, which can take a massive amount of time to process. Some tasks may take just a few minutes or hours while others may take several days to complete. In this paper, we design and develop a scalable architecture of API services for hosting ML models or DA functionalities in a production-grade deployment. The technologies of containerization and container orchestration, i.e., Docker and Kubernetes, have been employed to automate the deployment, scaling, and management of containerized ML or DA instances. To meet high-scale and high-availability requirements, the open-source message broker, i.e., RabbitMQ, is also used and containerized in Docker for scheduling multiple requests as task messages. These messages are then put into a task queue so that they will be processed later consecutively. Also, Nginx and Node.js with Express.js have been used and containerized as a web server and an API provider, respectively. We use a case-study of an intelligent system for processing documents about national research granting to validate our architecture.

Keywords—API service; Containerization; Queue Management; Machine Learning; Data Analytics

I. INTRODUCTION

An approach to achieve Artificial Intelligence (AI) is Machine Learning (ML), which uses algorithms to parse data, learn from them, and then make a determination or prediction about something benefiting the world. Data Analytics (DA) is an overlapping discipline of ML that mostly applies descriptive or inferential statistics to uncover a latent pattern or extract rich information from data. ML or DA is a complicated and time-consuming process. Just a single ML or DA process can spend many hours or days computing a result depending on the number of input data and the complexity of applying models. Therefore, most ML or DA processes are performed offline as all steps of the processes are carried out beforehand so that ML and DA services are ready at the time users submit their requests for computation. However, some of the processes are inevitable to be executed on-the-fly and processed simultaneously to serve many concurrent users. Due to an increasingly growing number of AI-oriented services, there has been a need to manage computing resources and a queue in processing the resource-intensive jobs as separate

threads of execution to any of the available resources and in an order manner.

Dependency management is another common problem in development and production environments. For instance, the setup of a development environment is not simple for novice developers or data scientists to install many dependencies to get projects run. Some ML models are fully functional only in Python version 3.6, but not in other versions. Furthermore, if ML models are previously built on Python version 3.6 and then deployed in a production environment where Python version 3.7 is installed, the models will be partially functional or, in the worst case, will not be able to be executed. Working together as a team among developers who use workstations with different operating systems and package managers is also troublesome. This is due to the fact that each workstation requires unique installation processes and often have a few different restrictions on the usage of dependent libraries.

In order to manage a queue of resource-intensive requests, and reduce the gap between development and deployment environments, we adopt the message queuing and containerization technologies. Software of message brokers [1] is used to create a Message Queue (aka. task queue) for accepting the requests from the application, also called producer¹. These messages are subsequently distributed among multiple workers (aka. consumers¹) that can flexibly be instantiated and destroyed as needed. Insides of a producer and consumers are composed of code, runtime and all their own dependencies (e.g., system tools, system libraries and settings.) Both of them must run reliably, immutably and independently from one computing environment to another. To achieve this, software of container-ization is used for encapsulating everything needed to run the application and services [2]. This encapsulation of all requirements for complete software to run is a standardized unit, so called a container, which is always ready for development and deployment.

To manage containers, container-orchestration needs to be deployed as Infrastructure-as-a-Service (IaaS). The orchestrator allows for automating the deployment, scaling, and management of distributed application. In our case, the ML modules are basically packaged up into container images and the orchestrator automatically allocate resources and spawn the instances from ML container images when there is a high need of more computing resources. Accordingly, this paper presents the scalable service architecture with request queuing for handling multiple

¹These two terms are often used in the context of message broker software.

resource-intensive tasks, e.g., those of ML and DA. Docker and Ku-bernetes are utilized as container and orchestrator systems, respectively. RabbitMQ is employed as a message broker system to organize submitted requests and distribute them to available ML container instances. Moreover, Nginx is used as a web server and Node.js with Express.js is used as a provider of RESTful API.

As our case-study, we implemented the intelligent system as a front-end application and the API as a back-end service provider to test our designed architecture in response to resource-intensive tasks. The services of the API exploit several techniques in AI, such as Natural Language Processing, Deep Learning, Topic Modeling and Information Retrieval for the analysis of Thai documents.

II. RELATED WORK AND TECHNOLOGY

In this section, we provide related theories and technologies which are the tools that are integrated in our scalable service architecture with request queuing.

A. Containerization

Containerization [3] is a form of operating-system-level virtualization that runs each application in a separated user space, called a container. In other words, a container is a lightweight method used to encapsulate an environment for an application so that it can work alone in its isolated container in any other machine with the same kernel-licensed operating system without considering its environment.

The right side of Fig. 1 shows that an operating system is not included in any container-like instance in the structure of containerization. The application of containerization reduces the duplication of resource usage in operating system but it requires the same underlying kernel-level operating system.

Docker [4] is one of the most popular sets of Platform-as-a-Service (PaaS) products that use operating-system-level virtualization to deliver containers. It is widely used to facilitate development and deliver the system due to the small consumption of resources in comparison with hardware virtualization. Besides, Docker does not work through a hypervisor but directly through the machine's kernel. Docker consists of four main components.

- Docker daemon is a service that runs on an operating system inside a Docker host and is responsible for communication between users and containers.
- Docker client is a program that receives commands directly from users, and forwards those commands to the Docker daemon to control resources.
- Docker registry is a highly scalable application in the server-side, which stores the Docker Image.
- Docker objects, i.e., Docker image and Docker container, are components that work together when Docker is running.

The Docker images are created to define the characteristics of a container. The docker container is a package that keeps the environment for running an application.

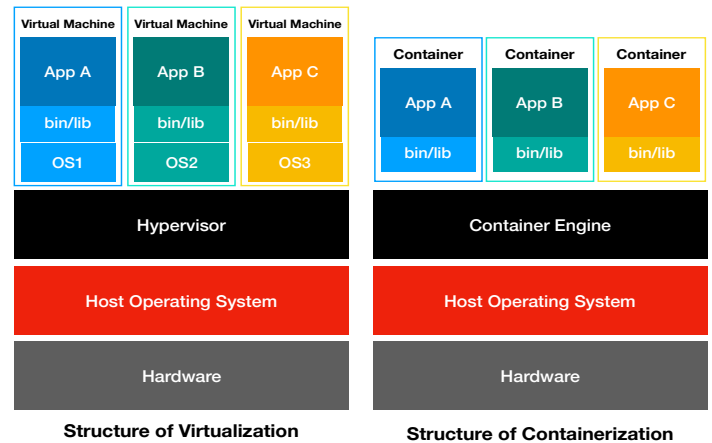


Fig. 1. Structures of Virtualization and Containerization

B. Container-Orchestration

Kubernetes [5] is an open-source container orchestration platform that automatically organizes the deployment, management and scaling for each application. It automates deployment and resource allocation and keeps containers alive by automatically starting up again after crash (auto self-healing).

C. Message Broker

Message broker or queue manager is a program that acts as an intermediary to receive messages and forward from a producer to a consumer. Message broker is also responsible for queuing messages and delivering the message to the desired destination. The popular message broker programs are, for instance, RabbitMQ, ActiveMQ and Kafka. Among message broker programs, RabbitMQ [1], [6] is the most widely deployed open source. This is because RabbitMQ is more sensitive to data transmission than others. This feature of RabbitMQ is a good option and appropriate for real-time processing [1]. Also, RabbitMQ supports Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT) and many other protocols.

D. RESTful API

Representational State Transfer (REST) is a form of web service that uses the HTTP Method (GET, POST, PUT, DELETE) [7]. Data is sent in the form of a JSON or XML file. RESTful API can send and receive data across platforms because it also uses a standard HTTP protocol widely used in today's websites. The benefit of using the RESTful API is that it is stateless. In other words, the system do not remember how the call occurred or was previously activated. By doing so, RESTful API makes a system easier to expand (higher scalable) and supports the increased usage. Caching is also possible and thus reduces the workload of the system. As a result, the overall speed of the system increases.

E. Model-View-Controller Architecture

Model-View-Controller (MVC) is conceptually the separation of concerns by organizing code according to their responsibilities. By applying this concept, application code is divided into three modules, i.e., model, view and controller.

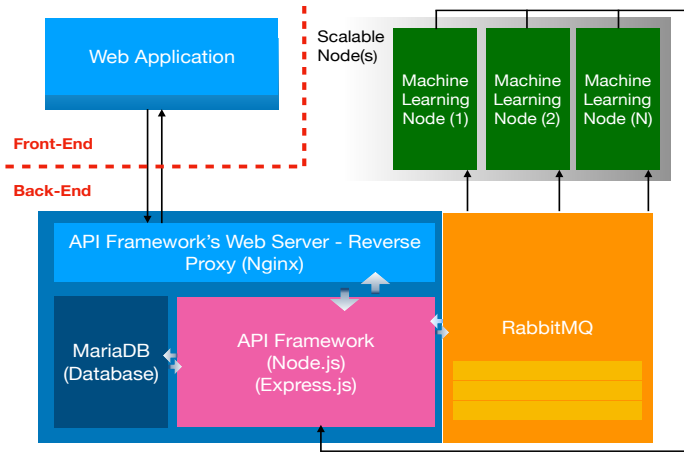


Fig. 2. Main components of our scalable system architecture

One module manages the data, another creates user interfaces for the application, and the other controls the application function. The MVC helps programmers separate code in a way that increases maintainability, and allows multiple developers to more easily collaborate and work together on different modules of code.

III. SYSTEM REQUIREMENTS AND DESIGN CONSTRAINTS

Our API is designed and developed to serve the requirements, from one of the main national funding agencies², of an intelligent system for processing documents about national research granting³. The system has three main features, including:

- 1) Uncovering thematic structures or topics hidden in a custom set of documents,
- 2) Finding keywords (can also be key phrases constructed by bi-gram or tri-gram) and then identifying keyword pairs of interest, and
- 3) Querying and ranking documents by a document example.

All features require a number of dependent libraries with exact versions, such as nltk, gensim, tensorflow, keras, bs4, etc., to be installed on development and deployment environments.

Served as a web application, the intelligent system invokes an API service by sending an HTTP request to process selected research documents. All three features are processed on-the-fly as per requested by users. For feature 1 and 2 processed simultaneously by the same request, the API spent, for instance, an estimated 3.30 hours to process 22 documents with average 260 pages long (min 124 and max 472 pages). For feature 3, it could take up to five hours for processing and indexing a hundred documents. Consequently, all services provided by our API are resource-intensive tasks due to the nature of the procedures in Natural Language Processing, Deep Learning, Topic Modeling and Information Retrieval. From the aforementioned reasons, the back-end API needs a scalable

²Thailand Science Research and Innovation (TSRI)

³Note that the details of machine learning processes running in the API are not the main contributions of this paper, and reserved confidential due to the law on protection of government information.

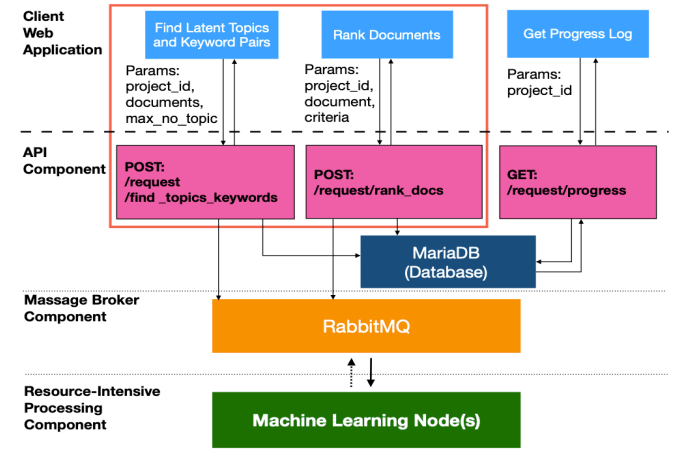


Fig. 3. Three API endpoints for routing requests to associated services.

architecture with containerization to separate it from the front-end, to queue and dequeue requests orderly, and to expand computing instance⁴ when needed.

IV. SYSTEM ARCHITECTURE

To fulfill the requirements of higher scalability and greater system availability, we design a scalable service architecture with request queuing for resource-intensive tasks (see Fig. 2). At the back-end, an Nginx web server acts as a center to accept requests, as configured by URL routing rules, and map them to their designed Restful endpoints. RabbitMQ receives many requests from the endpoints and puts them into a task queue so that any of them is not revoked. Database is also integrated in our architecture to store received requests so that they can be restored and forwarded to RabbitMQ when RabbitMQ does not respond. Besides, our Restful API also tracks the status of executed process returned from ML nodes, and then update the status for monitoring on the front-end Web application. After the process of ML is complete, the ML node send its result back to the API framework to be stored in the database and displayed on the front-end web application as well.

In addition, our system architecture is also designed for ease of deployment and operation as well as flexibility of ML node scaling. Thus, we make use of a Docker container to orga-nize the environment for development. From front-end to back-end, all components, such as Nginx, Node.js, MariaDB and ML node, are containerized in their own containers, which are later saved into image files, called “container images”. Apart from the original image of RabbitMQ that is maintained in Docker Hub, the container images of the other components are all stored in our repository in a Google container registry⁵ for simple deployment and scaling of additional ML nodes when existing nodes are full, i.e, reaching the maximum number of threads per node. Note that we currently fix the maximum number of threads per node to two threads. Kubernetes is also exploited to manage multiple containers and automatically scale the ML components by spawning ML instances from the registered container image of our ML node.

Fig. 3 elaborates our API endpoints, the center of the system architecture, with three URL request routes using

⁴Later called ML node, named after Machine Learning.

⁵<https://cloud.google.com/container-registry>

TABLE I. ELAPSED TIME TO FINISH FIVE CONCURRENT REQUESTS, AS PROCESSED BY THE DIFFERENT NUMBERS OF ML NODES.

#Request	#ML		
	1	3	5
5	5 hr 3 min	2 hr 57 min	2 hr 20 min

two common HTTP methods, i.e., POST and GET. The red square shows our main API endpoints, comprised of two route requests to particular ML processes. For the first endpoint with “**POST:/request/find topics keywords**”, it provides the services for features 1 and 2 (Section III), i.e., discovering latent topics and finding keyword pairs in a custom set of documents. Here, the API component saves all required parameters (i.e., project id, urls to download documents and the number of topics) to the database and forwards the request to RabbitMQ. Fig. 4 depicts an example of API testing of this endpoint by using Postman⁶. The second endpoint with “**POST:/request/rank docs**” offers the service for feature 3, ranking documents by their similarity to a query document. Similarly, all of its parameters are stored in the database before forwarded to the message broker component. The last endpoint, “**GET:/request/progress**”, is waiting for a request with a project id to send a progress log from the Database back to the web application.

V. API, SCALABILITY AND EFFICIENCY TESTING

We designed a preliminary experiment to test our API, and scalability and efficiency of our system architecture. For API testing, we used Postman to send POST and GET requests. For scalability and efficiency testing, we varied the number of maximum scaled ML nodes, i.e., 1, 3 and 5, and fixed only 1 thread for each node. As a controlled variable, the same POST request to process 22 documents was submitted five times (5 concurrent requests) simultaneously to the first endpoint via Postman. The specifications of a server, in which our back-end scalable service application was deployed, were also fixed by running a VM instance in Google Cloud. The configuration of the instance includes n1-standard-4⁷ (4 vCPUs, 15 GB memory) for machine type, Debian GNU/Linux 9 in 10 GB standard persistent disk for Boot disk and other default settings.

The result showed that our API had worked correctly as expected in queuing requests and calling ML nodes to perform any provided resource-intensive tasks. Moreover, our system could process multiple requests at once when there were multiple nodes available with less runtime than single node processing (See. Table I). That is, in our study the setting of 5 ML nodes outperformed the other two settings of 3 ML nodes and a single ML node by 20.90% and 53.80% faster, respectively.

VI. CONCLUSION

In this paper, we have introduced our scalable service architecture using container and container-orchestration technologies, message broker and RESTful API to fulfill the requirements from one of the main national funding agencies with higher scalability and serviceability. Kubernetes has been included in our architecture to manage multiple containers, Docker, for high scalability; and RabbitMQ has been used

⁶A collaboration platform for API development: <https://www.postman.com/> ⁷[https://cloud.google.com/compute/docs/machine-](https://cloud.google.com/compute/docs/machine-types)
types

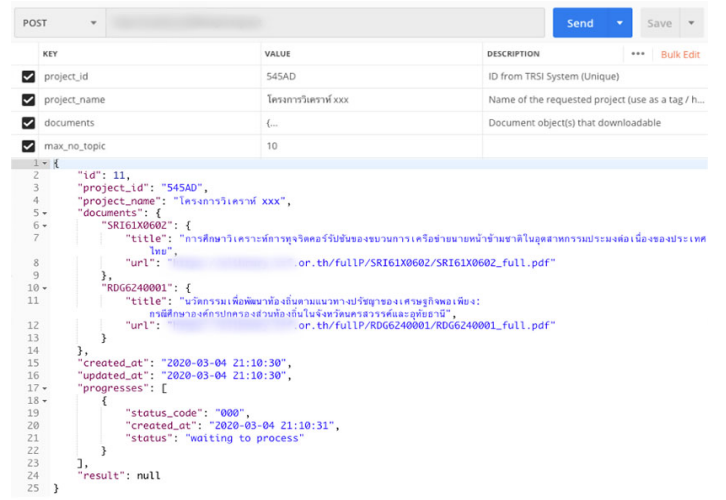


Fig. 4. An example of Postman interface, showing the test result of the first endpoint, “**POST:/request/find topics keywords**”, with input parameters and outputs returned in response to a request for running a ML process.

to manage queue and prevent any loss of requests. We have also implemented our RESTful API to be the center of our system architecture to pass through all the requests. Moreover, we have tested the capabilities of the architecture that we have designed by using Postman to act as a web application to send requests to and receive responses from the web server in our architecture. The results from our preliminary experiment have showed that our architecture was able to support the queuing for multiple requests and also to process concurrent multiple resource-intensive tasks with a lot less runtime than processing the tasks one by one.

ACKNOWLEDGMENT

This work was financially supported by the Thailand Science Research and Innovation (TSRI) under the project number RDG61Q0020, granted to T. Leelanupab to develop an intelligent system for processing documents about national research granting.

REFERENCES

- [1] P. Dobbelaere and K. S. Esmaili, “Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: In-dustry paper,” in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, 2017, pp. 227–238.
- [2] Y. Huang, K. cai, R. Zong, and Y. Mao, “Design and implementation of an edge computing platform architecture using docker and kubernetes for machine learning,” in *Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications*, 2019, pp. 29–32.
- [3] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support paas,” in *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610–614.
- [4] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, no. 239, p. 2, 2014.
- [5] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.
- [6] V. M. Ionescu, “The analysis of the performance of rabbitmq and activemq,” in *Proceedings of the 14th RoEduNet International Conference - Networking in Education and Research*, 2015, pp. 132–137.
- [7] L. Richardson and S. Ruby, *RESTful web services*. O’Reilly Media, Inc., 2008.