JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1133-1147 (2022)
DOI: 10.6688/JISE.202211_38(6).0003

A Microservices Orchestration Library based on PHP and RESTful API

WEN-TIN LEE[1,+], MENG-HSIEN WU[1], ZHUN-WEI LIU[1] AND SHIN-JIE LEE[2]
[1]*Department of Software Engineering and Management*
National Kaohsiung Normal University
Kaohsiung, 501 Taiwan
[2]*Department of Computer Science and Engineering*
National Cheng Kung University
Tainan, 701 Taiwan
E-mail: {wtlee[+]; 610877102; 611077104}@mail.nknu.edu.tw; jielee@mail.ncku.edu.tw

Microservices architecture has gradually become the primary consideration for the development of large software systems with scalability and flexibility. The orchestration and choreography patterns are provided to facilitate communications among microservices. Meanwhile, PHP is the programming language with the highest usage rate of the global web application servers. Nevertheless, there's still a lack of related PHP development resources in the field of microservices.

This work explores how to apply the service orchestration pattern to orchestrate the communications among microservice endpoints using PHP programming language. A set of service orchestration libraries, called Anser, are proposed based on PHP and RESTful API architecture. Developers can easily adopt the orchestration design pattern by using Anser to develop web applications based on microservices. Through performance evaluation, we show that Anser could facilitate the stability of microservices applications with lower error rates.

Keywords: microservices, microservices architecture, service orchestration, PHP, RESTful API, design pattern

1. INTRODUCTION

Microservices is an architecture style for developing an application with a collection of loosely coupled services that implement business capabilities. Adopting microservices architecture can bring several advantages: deliver and deploy complex applications continuously, better fault tolerance, easier to adopt new techniques, and easier to scale independently [1]. Applications developed based on microservices architecture are easier to maintain, deploy and scale than traditional monolithic applications. Each team responsible for developing services can be independent of other teams, and developers only need to focus on the functional code of each service. This allows large-scale applications to be developed in parallel, thereby significantly increasing the speed of software delivery.

Microservices architecture also brings challenges for the integration among plenty of small services as well as the selection of communication patterns. At present, most microservices communicate with each other via RESTful API based on lightweight protocols such as HTTP [2]. However, multiple small services mean more API needed to be managed. Some microservices that use API Gateway may need to obtain information provided by

Received December 2, 2021; revised January 19, 2022; accepted February 16, 2022.
Communicated by Shin-Jie Lee.
[*] This research was sponsored by Ministry of Science and Technology in Taiwan under grants No. MOST 110-2221-E-017-001-.

different service endpoints in one request. According to different system requirements, some features might need to complete complicated operations in one request which will involve communication between multiple service endpoints.

The orchestration and choreography patterns are provided to deal with the communication process among microservices [1]. Currently, Java is the mainstream programming language for microservices, sharing massive libraries and resources maintained by the open-source community. Compared with Java, PHP is the most used programming language for web applications in the world. Nevertheless, PHP and its open-source community rarely provide implementations and libraries of microservice-related design patterns.

This work designs and implements a set of libraries called Anser that can perform HTTP requests and process responses based on PHP and RESTful API architecture. Anser was named after the scientific name of wild-goose through the concept of formation, support, and organization in the geese theory to represent the process of constructing and orchestrating a large number of microservices in a distributed system. Developers can use the Anser orchestrator library to realize the orchestration design pattern quickly and build robust microservice systems using PHP to handle various application exceptions that may occur in HTTP communication by performing retry and recovery strategies.

The remainder of this paper is organized as follows: Session 2 presents background knowledge and related work. Session 3 describes the analysis and the design architecture of Anser libraries. Session 4 introduces the service and orchestration implementation methods. Session 5 conducts performance evaluations and explains the results. Finally, Session 6 concludes this study with future works.

2. RELATED WORK

Microservices are a software architecture inspired by service-oriented architecture, which has gained popularity in recent years [3]. Microservice architecture brings isolation into services, so the design patterns of microservices are different from the traditional ones. Jamshidi [4] pointed out that microservices can have better scalability and autonomy compared with monolithic applications. The relationship between services of monolithic applications is inseparable. When any service error occurs, it may cause delay and unavailability of the application. Moreover, microservices can provide high-performance services without losing communication reliability. More and more cloud-native architectures and containerization technologies based on microservices are available, making the deployment, expansion and management of microservices easier, flexible, fast, and effective [5]. Debagy and Martinek [6] mentioned that in some parallel testing scenarios, microservices surpassed monolithic architecture in performance and throughput. Richardson wrote that developers must deal with the additional complexity of creating a distributed system:

- inter-process communication and failure handling.
- requests span multiple services.
- writing test cases for interactions among services.

For the problems mentioned above, different design patterns and libraries have been developed. Next, we will discuss the structure and benefits of these patterns and libraries.

2.1 Service Orchestration and Service Choreography

In the microservices architecture, system functionalities are divided into individual small services. These services maintain their data and communicate through API invocations. Implementing complex business logic may be a process of operating different services in a specific coupling sequence, which requires the realization of orchestration and choreography design patterns. Gunawan *et al.* [7] implemented a service orchestration software framework based on Node.js. Fig. 1 describes the service orchestration pattern. All service execution sequences and exceptions are handled by the orchestrator.

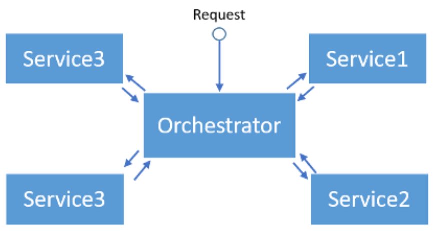


Fig. 1. Service orchestration.

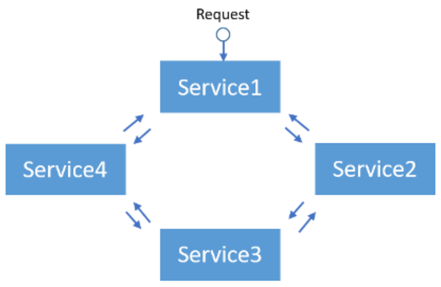


Fig. 2. Service choreography.

Fig. 2 depicts the service choreography pattern. The service will notify the next service to perform work after completing its own task. Usually, we will implement the message queue according to the rules of RabbitMQ or Kafka [1, 8]. Some researchers pointed out that implementing service choreography patterns makes it harder for the developers to maintain since it will scatter the business logic into the services [8, 9]. For simple business logic, choreography is more appropriate; For complex scenarios, although the performance of service orchestration is worse than the choreography, its characteristics make it a better choice for executing complex logic.

2.2 HTTP Protocol and RESTful API

HTTP is a common communication protocol, and the communication between the client and the server is usually implemented in the form of an HTTP API. The client will post a connection request, and the server will generate results in the form of HTML, JSON, or XML.

The representational State Transfer referred to as REST [7], has established an API design model for modern web applications. REST is an API design model that strictly conforms to the HTTP protocol, it focuses on describing resources by URL, and fully utilizes the standard of HTTP Request to take the method as a verb. If the HTTP API is implemented by the design concept of REST, it can be called a RESTful API. Currently, most microservices services use RESTful API to communicate with each other [1].

2.3 Service Orchestration Libraries

As shown in Table 1, we list libraries similar to Anser, which are PHP-based Vrata [10], Java-based SEATA [11], and Zuul [12]. Vrata is a PHP library that lacks functional support for retry strategies, filters, and error handling. SEATA is a Java library based on Spring Boot. Vrata and Zata use a markup language for service orchestration which means that developers cannot write more logic to express complex business processes. At the functional level, Anser provides the same functions as Zuul which is a mature Java microservice development library. Zuul expects developers to solve API composition and service orchestration problems by themselves, so they can only seek support from other libraries, or write from scratch, which will increase the complexity of programming.

Anser is developed in native PHP. Unlike other framework-based libraries, Anser can be imported into any framework, giving developers greater flexibility. For service requests, supporting error handling will give developers greater flexibility during development, log different HTTP errors, and return the correct error message. The retry strategy can greatly improve the success rate of communication with the microservice by requesting again after a period of time when the microservice is busy and the service is rejected. Filters give independent settings to specific connections before the actual request or perform some independent preprocessing after the request is successful, which is also an indispensable function in the microservice library. In terms of programming complexity, Anser provides many out-of-the-box functions in service orchestration. Using Anser, developers can avoid complex orchestration logic and state management, and can simply complete asynchronous service requests through a few lines of code.

Table 1. Comparison of microservice libraries.

	Anser	Vrata	SEATA	Zuul
Language	PHP		Java	
Framework	Native PHP	Lumen	Spring Boot	
Error Handling	Yes	None	Yes	Yes
Retry Strategy	Yes	None	Yes	Yes
Filter	Yes	None	Yes	Yes
Service Orchestration	Programming	Markup	Markup	Programming
Programming Complexity	Simple	None	None	Complex

3. ANSER MICROSERVICE ARCHITECTURE DESIGN

This work designs and implements a set of libraries based on PHP for the developers to implement the API composition pattern or service orchestration pattern conveniently.

Fig. 3 shows the legends used in the design model, Domain represents the scope of functions, Component represents a single component or function, Developer is the developer who uses Anser, Developer Component is the content component implemented by the developer, and External Component is the external component that uses Anser.

The Domain represents the function range, Component means a single component or function, Developer stands for the developer using Anser, Developer Component is the component implemented by developers, External Component describes the external components using Anser.

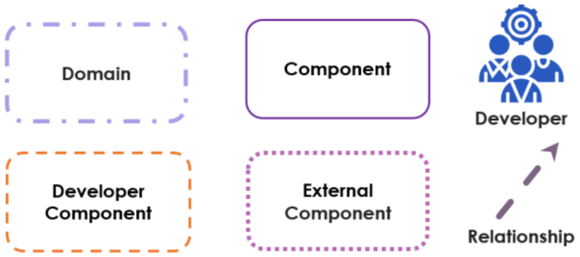


Fig. 3. Legend.

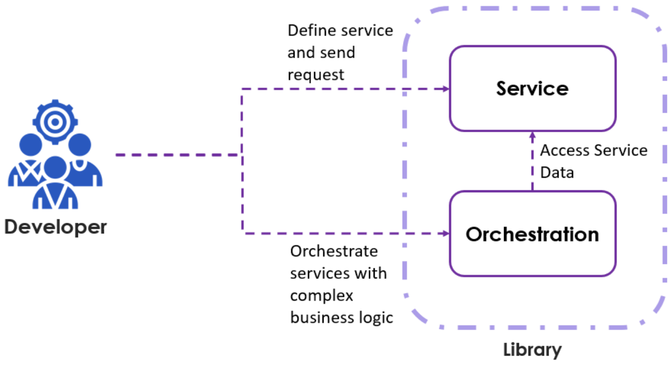

Define service and send request
Service
Access Service Data
Developer
Orchestration
Orchestrate services with complex business logic
Library
Fig. 4. Component diagram of Anser library.

Fig. 4 depicts the component diagram of the Anser library which includes two major development objectives ‘Service’ and ‘Orchestration’. Developers can use the Service component to define and implement their services. Through the Orchestration component, developers can orchestrate services with complex business logic.

3.1 Service Component Design

The Service component is used for implementing the microservice endpoint that may be called during program development, and meets the following functions:

1. Developers can create a list of available services by themselves.
2. Abstracts API invocation behavior with RESTful API
3. Able to define the behavior of each service when the request succeeds and fails.
4. Store the meaning data that services need to use after the server responds [2].
5. The service request supports a parallel connection.

Fig. 5 shows the detailed design of the Service component. Service provides four useable classes and one interface that needed to be implemented.

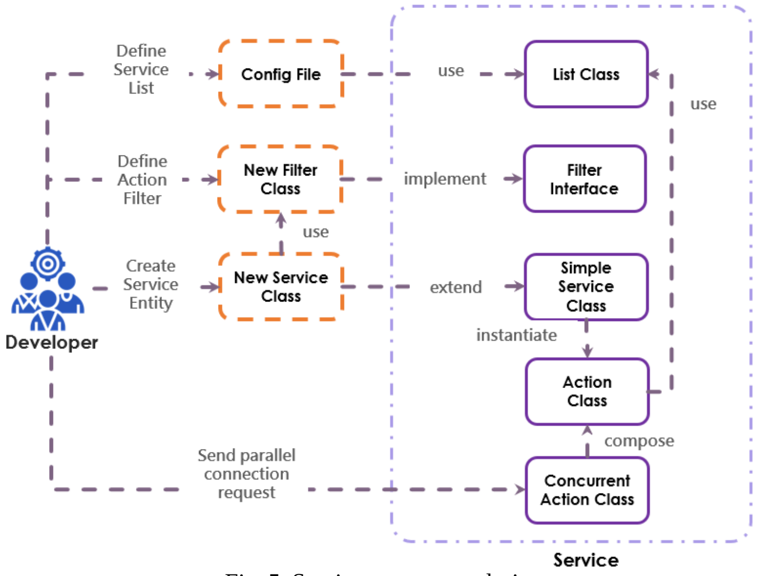


Fig. 5. Service component design.

- List Class: The only static class, developers can use this class to define usable service content.
- Action Class: An action is the smallest unit of a Service. It is responsible for abstracting the execution details of the HTTP connection, recording the execution result, allowing developers to intervene in the execution process, and allowing additional implementation of connection errors and processing logic when the execution is correct.
- Concurrent Action Class: If an HTTP connection needs to be a concurrent request, we need to instantiate this class and pass the Action to the request list.
- Simple Service Class: This is a basic class that integrates the above functions. By inheriting this class, the internal interface class can be defined for the APIs provided by the microservice endpoint.
- Filter Interface: Developers can define the logic to be processed before and after the Action execution.

3.2 Orchestration Component Design

Developers can organize a process to call multiple services through the Orchestration Component, and achieve the following functions:

- Developers can customize execution logic to support sequential execution and parallel processing.
- Developers can use the data generated during the runtime of the orchestrator in each process.

- Developers can define the output results after the execution of the orchestrator by themselves.

Fig. 6 shows the detailed design of the Orchestration component which provides two classes.

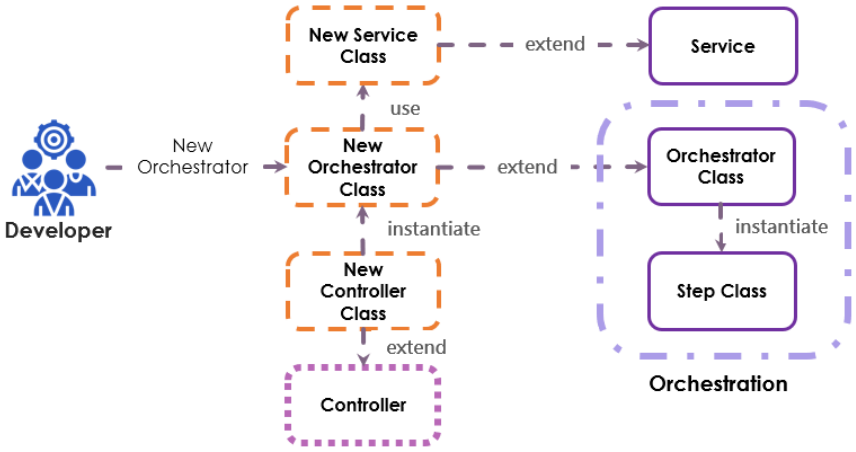


Fig. 6. Orchestration component design.

- Orchestrator Class: An abstract class that provides the methods needed by the orchestrator when executing across multiple services. After extending this class, abstract methods must be implemented to define the execution details of the orchestrator.
- Step Class: This class does not need to be inherited or implemented. It will be instantiated by the orchestrator and used to manage the execution of the complex orchestration process.

The New Controller Class extends the Controller class which is based on native PHP. We expect that developers will use this library in the PHP frameworks, hence the library developed by this work is based on native PHP, which can be instantiated by any external class and operate normally.

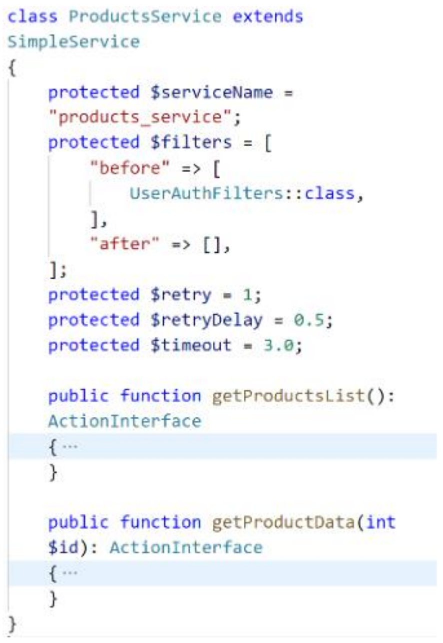
4. ANSER MICROSERVICE ARCHITECTURE IMPLEMENTATION

The Anser library brings developers many out-of-the-box classes and methods. Developers only need to write public methods and make detailed settings for the endpoint resources of microservices. These settings include endpoint location, request method, response parsing method, *etc.* Anser expects microservice developers to easily implement service orchestration in a programming way, and further maintenance, debugging and testing can be guaranteed more smoothly. We demonstrate how to use Anser to implement service orchestration in the remainder of this section.

4.1 Service and Action Implementation

Fig. 7 shows the implementation of SimpleService. ProductsService extends the SimpleService class to implement service content. In the ProductsService class, we map the resources provided by the microservice endpoints, offer the resources before and after the HTTP request, and set the number of retries, intervals, and timeout when the request fails. The public methods represent the detailed configuration of the endpoint's resources and usually return Action objects.

Fig. 8 describes how the Action object is used. The action object will request a RESTful API using the HTTP protocol to retrieve a list of products. In this example, developers can pass the product list to callback functions to handle the logic when the request succeeds or fails.

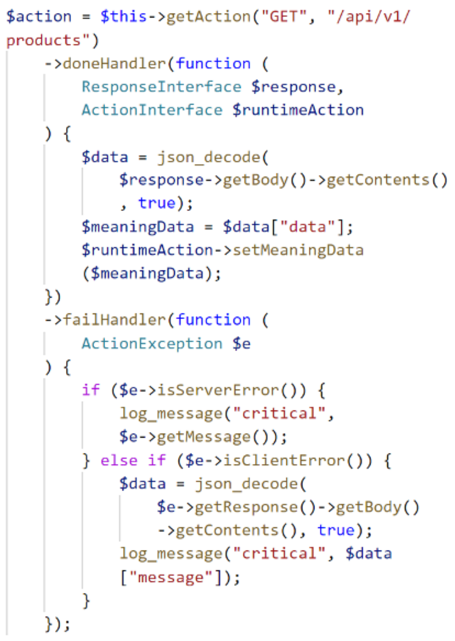


```
class ProductsService extends
SimpleService
{
    protected $serviceName =
    "products_service";
    protected $filters = [
        "before" => [
            UserAuthFilters::class,
        ],
        "after" => [],
    ];
    protected $retry = 1;
    protected $retryDelay = 0.5;
    protected $timeout = 3.0;

    public function getProductsList():
    ActionInterface
    {…
    }

    public function getProductData(int
    $id): ActionInterface
    {…
    }
}
```

Fig. 7. Simple service implementation.



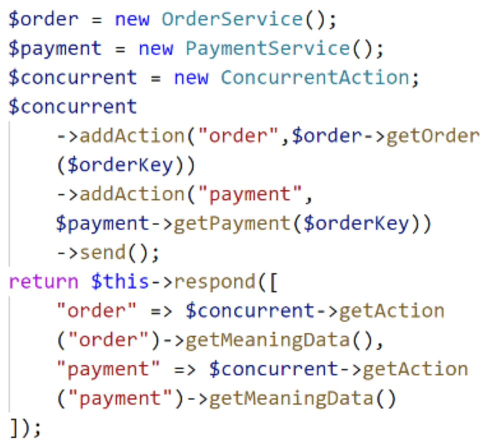
```
$action = $this->getAction("GET", "/api/v1/
products")
    ->doneHandler(function (
        ResponseInterface $response,
        ActionInterface $runtimeAction
    ) {
        $data = json_decode(
            $response->getBody()->getContents()
            , true);
        $meaningData = $data["data"];
        $runtimeAction->setMeaningData
        ($meaningData);
    })
    ->failHandler(function (
        ActionException $e
    ) {
        if ($e->isServerError()) {
            log_message("critical",
            $e->getMessage());
        } else if ($e->isClientError()) {
            $data = json_decode(
                $e->getResponse()->getBody()
                ->getContents(), true);
            log_message("critical", $data
            ["message"]);
        }
    });
```

Fig. 8. Using action object.

4.2 Concurrent Action and Orchestration Implementation

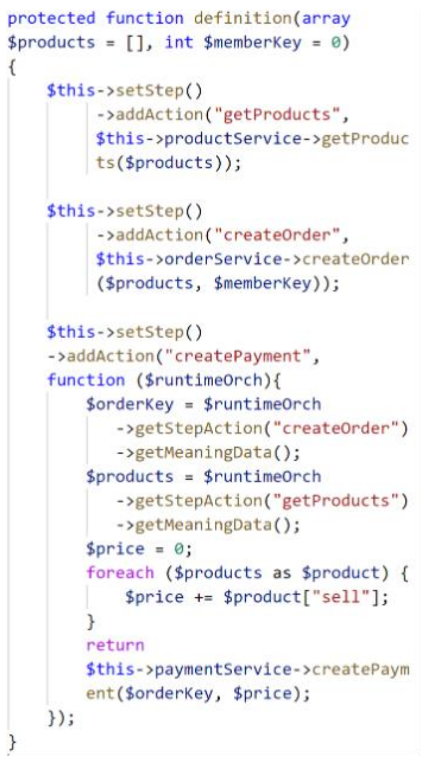
Fig. 11 describes that developers can pass multiple action objects into the Concurrent Action object to achieve parallel requests for multiple endpoints. ConcurrentAction object integrate and return the responses after the request is successful.

Anser provides an abstract class Orchestrator for developers to quickly implement service orchestration pattern. Fig. 12 shows that developers can define execution processes involving multiple microservices and with orders. In the example, you can find that each Step is composed of several Actions. If more than one Step is defined, then these Steps will be executed sequentially in the orchestrator. The Step can also retrieve other Step's execution results during execution. You only need to pass in the corresponding callback function to obtain the required Step Action through the Orchestrator object.



```
$order = new OrderService();
$payment = new PaymentService();
$concurrent = new ConcurrentAction;
$concurrent
    ->addAction("order",$order->getOrder
    ($orderKey))
    ->addAction("payment",
    $payment->getPayment($orderKey))
    ->send();
return $this->respond([
    "order" => $concurrent->getAction
    ("order")->getMeaningData(),
    "payment" => $concurrent->getAction
    ("payment")->getMeaningData()
]);
```

Fig. 9. API composition pattern implementation.



```
protected function definition(array
$products = [], int $memberKey = 0)
{
    $this->setStep()
        ->addAction("getProducts",
        $this->productService->getProduc
        ts($products));

    $this->setStep()
        ->addAction("createOrder",
        $this->orderService->createOrder
        ($products, $memberKey));

    $this->setStep()
    ->addAction("createPayment",
    function ($runtimeOrch){
        $orderKey = $runtimeOrch
            ->getStepAction("createOrder")
            ->getMeaningData();
        $products = $runtimeOrch
            ->getStepAction("getProducts")
            ->getMeaningData();
        $price = 0;
        foreach ($products as $product) {
            $price += $product["sell"];
        }
        return
        $this->paymentService->createPaym
        ent($orderKey, $price);
    });
}
```

Fig. 10. Service orchestration implementation.

5. PERFORMANCE EVALUATION

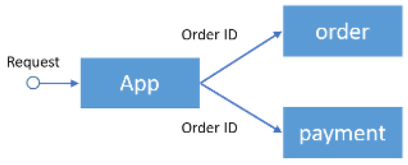
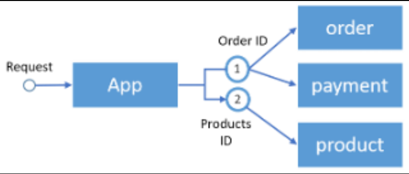
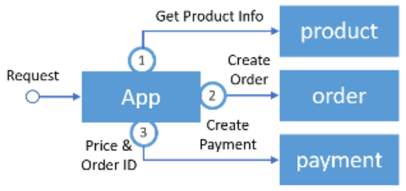
For performance evaluation, we designed three independent microservices with independent databases. As shown in Table 2, all microservices are written in PHP and use PostgreSQL as the database for data storage. The three microservices were deployed on three independent hosts located in the same LAN using Docker. All microservices expose a set of RESTful APIs, which serve the CRUD (add, read, edit, delete) requirements of resources such as Order, Payment, and Product. The hardware used for the experiment is a 2.3GHz, 8-core Intel Core i7 processor, and 16 GB 2667 MHz DDR4 memory.

We compared Anser with a PHP open-source Gateway 'Vrata' which is written based on an open-source framework called Lumen. Vrata provides proxy, composition, and orchestration functions for multiple microservices. The orchestration details of services must be defined in the JSON data format. In the performance tests, we designed three cases to simulate the actual requirements of writing microservices and tested the performance under these conditions. Table 3 shows the three test cases designed for the performance testing of service orchestration.

Table 2. Service for testing.

Service Name	Functions
Product	Product description, price, inventory
Order	Stored order number, commodity, amount
Payment	Record payment amount and info

Table 3. Test cases.

Cases	*Execution Details*
Order Overview	
	1. Use the parallel connection to obtain data from the Order and Payment micro-services by passing in the Order ID. 2. Return the results after combining the responses of the two microservices.
Order Details	 Request App Order ID 1 2 Products ID order payment product
	1. Get the Order and Payment data from the parallel connection after passing in the ID. 2. Then request product details from the Product microservice with the 'Products ID' returned in the previous step. 3. Combine the responses of all microservice and return them.
Create Order	 Request App Get Product Info 1 Create Order 2 Create Payment 3 Price & Order ID product order payment
	1. Obtain prices of all products from product microservice after passing in the product ID array, the required quantity, and the user ID. 2. Pass the user ID, product array, and purchased quantity to the order microservice to create a new order and get the order number. 3. Pass in the order number and checkout amount to payment microservice after calculating the checkout amount based on the product sales price and required quantity in the first step, 4. After confirming the data is written, the created order number will be returned.

We will use the Anser library to implement the execution details defined in the test cases. All experiments will simulate actual user connections through JMeter, and restart all servers' docker containers after every simulated connection is completed.

5.1 Order Overview Evaluation

According to the results shown in Tables 4 and 5, when the server is facing high loads, Vrata provides faster execution speed but also has a higher error rate of 11.17%. Although Anser performs slower in executing speed, it turns out that it can handle all connections correctly.

Table 4. Order overview: 10000 times sampling test.

Thread	1000	Interval	1 sec
Loop	10	Sample amount	10000

	Anser Library	Vrata
Avg. (ms)	5427	3626
Error rate	0.0%	11.17%
Throughout (sec)	173.7	191.4

Table 5. Order overview: 5000 times sampling test.

Thread	500	Interval	1 sec
Loop	10	Sample amount	5000

	Anser Library	Vrata
Avg. (ms)	2674	2328
Error rate	0.0%	8.28%
Throughout (sec)	172.4	184.3

The request results of the microservices implemented using Anser will be processed by PHP's "json_decode()" method; while Vrata turns out to directly combine the microservices' responses into the final output. Therefore, in the face of high-load requests, it will take more time to process the decoding and transcoding of the responses while implementing the Anser library.

5.2 Order Details Evaluation

According to the results in Tables 6 and 7, Anser's execution speed will become closer to Vrata for complex service compositions in a low-load environment. Until the sampling rate reaches 2000, the execution result of the Anser library becomes better than Vrata.

In addition, Vrata still has a certain degree of error rate in a high load environment. Such an error rate will lead to instability in service provision. The Anser library can provide more stable performance while performing complex service orchestration. It can provide accurate and error-free services in a high-load environment.

Table 6. Order detail: 5000 times sampling test.

Thread	500	Interval	1 sec
Loop	10	Sample amount	5000

	Anser Library	Vrata
Avg. (ms)	7323	6708
Error rate	0.0%	5.32%
Throughout (sec)	64.3	67.2

Table 7. Order detail: 2000 times sampling test.

Thread	200	Interval	1 sec
Loop	10	Sample amount	2000

	Anser Library	Vrata
Avg. (ms)	2802	2855
Error rate	0.0%	5.32%
Throughout (sec)	65.9	65.0

5.3 Create Order Evaluation

Tables 8 and 9 show the performance of different microservices to get and write data at the same time during service orchestration. When facing high load requests, although the processing time of the Anser library is higher than that of Vrata, it has a stability of zero error rate. On the contrary, regardless of high load or low load, Vrata cannot guarantee the stable execution of the order creation task. For distributed systems, the stability of creations and modifications is more important than performance.

Fig. 11 shows the error rate of the Anser library and Vrata under different cases of the loading test with 5000 samples. Anser can maintain zero error stability in all test cases. After checking the server log, we found that in either case, Vrata will first encounter the problem that the number of connections exceeds the upper limit of the server software and the connection is rejected. Therefore, the rest of the errors are triggered by server execution timeout, and at the same time, the microservices work normally. It can be concluded that Vrata has flaws in task scheduling under high load and cannot effectively retry the failed services to ensure the correct execution of requests.

The Anser library developed in this work can provide PHP developers with new choices when implementing the microservice architecture, and brings the following benefits:

- Compatible with different PHP software development frameworks.
- Provide more stable execution performance compared with similar solutions.
- Ensure that the services run normally under high concurrent requests with low error rates.

Table 8. Order creation: 10000 times sampling test.

Thread	1000	Interval	1 sec
Loop	10	Sample amount	10000

	Anser Library	Vrata
Avg. (ms)	8719	5612
Error rate	0.0%	18.27%
Throughout (sec)	108.7	117.9

Table 9. Order creation: 5000 times sampling test.

Thread	500	Interval	1 sec
Loop	10	Sample amount	5000

	Anser Library	Vrata
Avg. (ms)	4232	3831
Error rate	0.0%	6.90%
Throughout (sec)	109.7	115.1

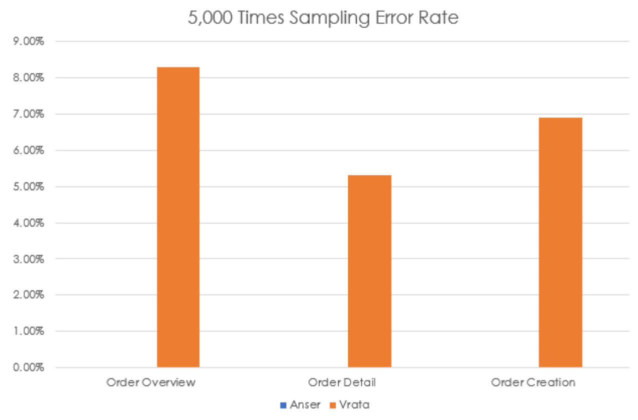


Fig. 11. Bar graph of 5000 sampling error rate.

6. CONCLUSION

To solve the maintenance problems of large-scale applications on complex architectures, it is imperative to adopt microservices for large-scale project development. This work proposes a microservice solution based on the PHP programming language called Anser (https://github.com/SDPM-lab/Anser-Action), which provides PHP developers with new options for microservice implementation. Developers do not need to migrate to unfamiliar programming languages to implement microservices. In the past, using PHP to communicate with other endpoints through the HTTP protocol was a very tedious task. Using the Anser library, developers can implement functions such as the integration of multiple services and complex service orchestration on the premise of familiar language and high performance.

This study proposes a service orchestration library available in PHP regarding microservice architectures implemented in different languages. In terms of performance evaluation, PHP currently does not have a representative microservice solution, so this study only compares Vrata with Anser. At the same time, PHP must rely on server software. The choice of different server software and caching mechanisms will also make the library have different performance. There should be more in-depth experimental planning for different server software. There should be more in-depth experimental planning for different server software. If the software architecture adopts the microservice architecture, it will face the challenge of data consistency in the distributed database [1]. Such challenges must rely on developers to solve the data consistency in the program. At present, there are also related design patterns as well as libraries to support developers in maintaining data consistency in microservices [7, 13, 14]. In the future, we will propose simple and easy-to-use solutions to solve the challenges of data consistency under the service orchestration.

REFERENCES

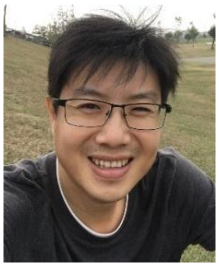
1. C. Richardson, *Microservices Patterns: With examples in Java*, Manning Publications, NY, 2018.

2. K. Malyuga, O. Perl, A. Slapoguzov, and I. Perl, "Fault tolerant central saga orchestrator in RESTful architecture," in *Proceedings of IEEE 26th Conference of Open Innovations Association*, 2020, pp. 278-283.
3. N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, eds., Springer International Publishing, Cham, 2017, pp. 195-216.
4. P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, Vol. 35, 2018, pp. 24-35.
5. A. Warke, M. Mohamed, R. Engel, H. Ludwig, W. Sawdon, and L. Liu, "Storage service orchestration with container elasticity," in *Proceedings of IEEE 4th International Conference on Collaboration and Internet Computing*, 2018, pp. 283-292.
6. O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *Proceedings of IEEE 18th International Symposium on Computational Intelligence and Informatics*, 2018, pp. 000149-000154.
7. G. F. Gunawan, J. F. Palandi, and Subari, "Redesigning CHIML: Orchestration language for chimera-framework" in *Proceedings of the 3rd International Conference on Informatics and Computing*, 2018, pp. 1-7.
8. C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," *International Journal of Advanced Computer Science and Applications*, Vol. 9, 2018, pp. 18-22.
9. R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, Vol. 2, 2002, pp. 115-150.
10. PoweredLocal, "Vrata," https://github.com/PoweredLocal/vrata, 2022.
11. Alibaba, "SEATA," https://seata.io/en-us/, 2022.
12. Netflix, "Netflix Zuul," https://github.com/Netflix/zuul, 2022.
13. H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1987, pp. 249-259.
14. X. Limón, A. Guerra-Hernández, A. J. Sánchez-García, and J. C. P. Arriaga, "SagaMAS: A software framework for distributed transactions in the microservice architecture," in *Proceedings of the 6th International Conference in Software Engineering Research and Innovation*, 2018, pp. 50-58.

Wen-Tin Lee (李文廷) received his Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 2008. Lee is currently an Associate Professor in the Department of Software Engineering and Management at National Kaohsiung Normal University. His research interests include software engineering, service-oriented computing, and deep learning.

Meng-Hsien Wu (吳孟賢) received his master degree in the Department of Software Engineering and Management at National Kaohsiung Normal University, Taiwan, in 2021. His research interests include software engineering, microservice architecture, web programming, and service computing.

Zhun-Wei Liu (劉峻維) is currently a master student in the Department of Software Engineering and Management at National Kaohsiung Normal University, Taiwan. His research interests include software engineering, microservice architecture, DevOps, web programming, and container technology.



Shin-Jie Lee (李信杰) is an Associate Professor in Computer and Network Center at National Cheng Kung University in Taiwan and holds joint appointments from the Department of Computer Science and Information Engineering at NCKU. His current research interests include software engineering and service-oriented computing. He received his Ph.D. degree in Computer Science and Information Engineering from National Central University in Taiwan in 2007.