Redesigning CHIML: Orchestration Language for Chimera-Framework

Go Frendi Gunawan
STIKI Malang
Malang, Indonesia
Email: frendi@stiki.ac.id

Jozua Ferjanus Palandi
STIKI Malang
Malang, Indonesia
Email: jozuafp@stiki.ac.id

Subari
STIKI Malang
Malang, Indonesia
Email: subari@stiki.ac.id

***Abstract*—Component-Based Software Engineering (CBSE) has been proven to be quite effective to deal with software complexity. Nowadays developers prefer to build micro-services rather than a single monolithic application. Several SOA (Service Oriented Architecture) approaches like HTTP/REST API, CORBA, and BPEL are commonly used by developers. Some of those solutions are built under assumptions that the developers are either building the services from scratch or able to create abstraction layer for the pre-existing services. In most cases the assumptions are true. However, there are cases when developers prefer to keep the architecture as simple as possible without any need to build additional abstraction layers. For example, when they work with the mini-embeded system.**

Previously, a YAML based orchestration language was developed for Chimera-Framework (A language agnostic framework for stand-alone and distributed computing). However, we observe some weak points in the language. In this paper, we refine the orchestration language in order to let developers accessing pre-existing services without any need to build another abstraction layer.

Keywords—Chimera, CBSE, Orchestration Language

I. Introduction

Software development is a very interesting topic. The way people developing softwares is changing as new paradigms emerged. In turns, software development also affecting the culture. It changes how people interact with each other as well as how they interact with computers.

As software becomes more and more complex, building and maintaining software is also become harder. Various approaches have been attempted in order to make the processes easier.

Several frameworks like Rails [1], Django [2], Laravel [3], etc are focusing on how to make codes clean, separated, and reusable. In turn, this make software development and maintenance easier. These frameworks quickly gained popularity among software developers.

However, despite on the clear advantage of those frameworks, they also have several disadvantages. Since the frameworks were built on top of specific programming language, integration to different programming languages is quite challenging. For example, in order to build software by using Rails, developers should code in Ruby. The same is also true for Django, which is built on top of Python, and Laravel which is built on top of PHP. This situation is known as vendor-lock-in [4]. And in some cases, it can affect maintainability as well as further development in a bad way.

Thus, a more technology agnostic approach is needed in order to overcome a vendor-lock-in problem. Recently, SOA and micro-services are gaining popularity. Those approaches let developers to focus on small components known as services rather than complex monolithic software. A service usually represents a single resource or business process. The services can be independent of each other or aware of each other. Despite of the advantages and popularity, SOA and micro-services are prone to concurrency problem [5].

One important aspect in SOA/micro-services is how a developer can compose independent services to work together. There are two common approaches to compose services. The process to compose several services that are independent of each others is named orchestration, while the process to compose several services that aware of each other is named choreography. Orchestration require one central controller in order to manage the services [6].

Orchestration has a very long history. The earliest implementation of orchestration was Unix Pipe mechanism [7]. This mechanism is still relevant and used by Unix/Linux users.

Unix Pipe mechanism is not the only implementation of software orchestration. Remote Procedure Call (RPC) is also used for bigger projects. The commonly used RPCs are JSON-RPC [8] and XML-RPC [9]. In 2015, Google also introduces GRPC [10].

Beside Unix Pipe and RPC, several other implementations were introduced by independent companies and consortiums. OMG introduced CORBA on 1991, while OASIS introduced BPEL in 2001. Some developers also implement their own HTTP/REST API implementation. In 2016, Feilhauer and Sobotka creating a framework named DEF which is focusing on parallel execution [11]. In 2017, we also conduct a research to develop another language agnostic framework named Chimera-Framework [12].

Compared to the monolithic frameworks like Rails and Django, these orchestration mechanisms are more scalable and technology agnostic. This means that the developers can choose the best technology stack to build their components/services.

Among those orchestration mechanisms, Unix pipe and Chimera-Framework are the only ones that are supporting but not limited to network protocol. Typically, Service Oriented

Architectures (like CORBA, BPEL, or DEF) are correlated to a distributed architecture, but this is not always the case. For example, a developer might only need to pass data from *R* to *Python* on the same machine.

Compared to the existing mechanisms, Chimera-Framework focused on how to let developer effort as minimal as possible. The orchestration components in Chimera-Framework don't have to be HTTP aware. Even an old UNIX utility like *date* or *cat* can serve as components [12]. But in the case of distributed architecture is required, the same components are still going to be usable.

In this research, we are focusing on improving the orchestration mechanism in Chimera-Framework. The orchestration language is named CHIML (Chimera Markup Language) which is a superset of YAML [13].

CHIML is designed to be readable, compact, and intuitive, yet it has several weaknesses:

- CHIML is slow because the framework invokes *vm.runInNewContext* for every statements.
- CHIML support for parallel computing is limited, especially if the parallelization is data oriented.
- CHIML use it's own HTTP protocol for network communication, which is uncommon.

In this research, we are going to solve this two problem by redesigning CHIML.

II. Research Question

In order to have a clear direction in our research, we are focusing on these four questions:

- How to improve CHIML execution speed.
- How to make CHIML supporting data-oriented parallelization.
- How to implement a more common protocol for network communication.
- How to keep CHIML readable, compact, and intuitive.

III. Literature Survey

A. Orchestration and Choreography

In 1975, Frank DeRemer and Hans Kron wrote a paper about programming-in-large and programming-in-small. Programming-in-large is a concept to write a software that consists of various modules that are possibly written by different people [14].

The concept of programming-in-large is tightly coupled to orchestration and choreography. In orchestration, the components are controlled by a single controller, while in choreography, the components are aware of each other. In some cases, orchestration and choreography can be used together. Developer might create a choreography of components where the components are the orchestration of smaller components [6].

B. SOA and Micro-service

SOA and micro-service are similar architecture concept. The core philosophy of these two paradigms is to split the big problem into small problems. The term SOA is usually being used when people talk in a bigger scope, while micro-service is usually referred to a smaller scope. When services are composed and used internally it is micro-service. But when the services exposed to external system, it is SOA. Although the distinction can be unclear, people usually agree that SOA might contain micro-services [5].

From the technical point of view, SOA and micro-service might involve either orchestration or choreography. And no matter which one is being used, a protocol for passing message among services is required.

C. HTTP/REST API

HTTP API is a quite common protocol. The architecture is also relatively simple. There are web services and a clients. The client sends a request to the web service and the web service reply with a response. The response data is usually in JSON/XML format. SOA components are commonly exposing HTTP API endpoint. OMDB for example, provide an API to search for movie database.

On the other hand, REST (Representation State Transfer) is a more strict implementation of HTTP API. It was introduced and defined in 2000 by Roy Fielding in his doctoral thesis [15]. Fielding focusing on how a URI should represent an object, while HTTP verb serve as it's method. If the endpoint is fully adapting REST specification, it is called *RESTful web service*.

D. JSON-RPC, XML-RPC, and GRPC

RPC stands for Remote Procedure Call. The focus of RPC is to let developer invoke procedures in remote computer as easy as they are a local procedure. JSON-RPC and XML-RPC work in the same manner, only the data exchange format is different. As the names implied, JSON-RPC use JSON [8], while XML-RPC use XML [9]. First, the client send a request to the server. The request contains the name of the method and the parameters. The server then replied with the return value.

As XML is more verbose than JSON, XML-RPC is also need more bandwidth compared to JSON-RPC. And as XML and JSON are text format, binary data should also be encoded in text format. For example, an image might need to be encoded in base-64 format.

To overcome the need to encode data into text format, Google create another protocol named GRPC (Google Remote Procedure Call). GRPC supporting several programming languages including Python, Javascript, Java, and PHP. In order to use GRPC [10], developers have to create stub skeleton (similar to IDL in CORBA, BPEL, or EJB).

IV. CHIML

A. Improving Speed

After doing several investigations, we find that there are two reasons why CHIML is slow:

- CHIML invoke *vm.runInNewContext* for every statements.
- Node.Js need to check all corresponding directories in order to resolve imported module path.

In order to solve the first problem, we rewrite CHIML interpreter, and build a transpiler instead. So, rather than invoking *vm.runInNewContext* for every statements, we translate the whole CHIML script into valid Javascript. Thus, we only need to call *vm.runInNewContext once*. This approach let us explore another possibility like really build a runnable Javascript that theoretically is runnable and decoupled from Chimera-Framework.

As for the second problem, many developers has tried to overcome the problem by overriding Node.Js native *require* and build cached to speed up the *module-resolve* process.

B. Supporting Data-Oriented Parallelization

In the previous version, CHIML already supporting parallelization by using *parallel* keyword. However, this mechanism works with the assumption that the developers already know how many parallel process is needed. Since some time, the parallelization is based on user inputs/fetched data, this mechanism becomes unusable in those cases.

To solve this problem, we decide to support *map, filter, and reduce*. These concepts are pretty common in *functional programming*, and it is pretty useful to transform array. Behind the scene, we translate the *map, filter, and reduce* into *Promise.all()* to make it truly parallel.

C. More Common Network-Based Data-Exchange Protocol

In the last years, many new protocols like GraphQL emerged. Some argue that GraphQL would be the new standard protocol replacing REST-API. However, this is not true. GraphQL is just another wrapper layer encapsulating several REST endpoints.

Furthermore, we think REST is basically a single endpoint that encapsulates several RPCs into a single resource. Thus, we can say that the building block of network-based data-exchange communication is RPC.

Since CHIML is based on JavaScript, and JavaScript is close to JSON, we decide to use JSON-RPC for exchanging data over network. Beside JSON-RPC, the plain HTTP-Request is also supported. Supporting plain HTTP-Request is important, since it allow developers to build their own custom protocols whenever they need to.

By using those protocols, we believe Chimera will be more useful since it can talk to different systems seamlessly.

D. Readable Language

There are several metrics to measure readability. In 1975, Kincaid introduces a formula to measure text readability [16]. Even though the formula is commonly used for measuring text readability, some people argue that programming language should have different kind of metrics.

In 2010, Buse and his team introduced a new metric to measure the readability of programming language [17]. The metric is basically a modified version of Kincaid's formula.

We try to use these metrics in order to compare CHIML with *Python* and *Javascript*.

E. Semantic Design

The brief overview of CHIML semantic is presented at Listing 1 as Typescript interface.

Listing 1. CHIML Semantic Design
```
// Structure of a 'command'
interface CommandObject{
  ins : string | string[],
  out: string,
  vars: {[key: string]: any},

  if: string = "true",

  map: string,
  filter: string,
  reduce: string,

  into: string,

  accumulator: string,

  do: command | command[],
  parallel: command[],

  while: string = "false",
};
```

In order to support both, programming-in-large, and programming-in-small, we put branch and loop control in the semantic (*if* and *while*).

Map, filter, and reduce are mutually exclusive. Developer can only use one of them, along with *into* key. *Accumulator* is only used in *reduce* operation.

Since we are focusing on how to make the language more compact, we also support several short-hands so that a single process can be written in a single line.

For example, the process in Listing 2 can also be written as in Listing 3

Furthermore, since developers might use to write left-hand assignment in mainstream programming languages (i.e: *result = function(param1, param2)*), we think reverse arrow in CHIML can make it more intuitive. Thus, Listing 3 can also be written as 4.

Listing 2. CHIML Program Example
```
ins:
  month
  year
out: calendar
do: cal
```

Listing 3. CHIML Program Example (Short)
```
(month, year) -> cal -> calendar
```

Listing 4. CHIML Program Example (Short-Reversed)
```
calendar <- cal <- (month, year)
```

The nature of CHIML that can be used for programming-in-large and programming-in-small allows the developer to quickly create prototypes. For example, to get the content of page-1 to page-10 of a website, the developer can do the iteration as displayed in Listing 5

Listing 5. CHIML Iteration
```
ins: url
out: html
do:
  - i <— 1
  - while: i < 11
    do:
      - (url + '?page=' + i) -> curl -> htmlFragment
      - (html + htmlFragment) —> html
      - i <— (i+1)
```

We also take further analysis and conclude that for similar cases, it is better to do the task in parallel, so that the first request doesn't block the later requests. This can be perfectly achieved by using *map* mechanism as shown in Listing 6:

Listing 6. CHIML Map Feature
```
ins: url
out: html
vars:
  indexes: [1,2,3,4,5,6,7,8,9,10]
do:
  - map: indexes
    into: responses
    ins: index
    do:
        - (url + '?page=' + index) -> curl ->
            response

  - |(responses, '') -> {(array) => array.join("")}
      -> html
```

Map is more efficient since under the hood, each iteration will be done in a non-blocking manner.

F. Implementation

In order to parse and execute CHIML script, we have already build a parser to evaluate the script and execute it on the fly.

V. Experiment

A. Problem

For the test-case, we provide three HTTP API endpoint exposing three tables, *genres*, *books*, and *authors*. These HTTP endpoints are assumed as a black-box system. The only way to interact with those APIs is by sending HTTP GET request.

To select all the data from each end-point, the user can send GET request to *http://localhost:3000/[table-name]*. The user can also filter the data by adding *field-name* and *value* as GET parameter. For example, if the user want to select all books written that have genreId = 5, the user should access this url: *http://localhost:3000/books?genreId=5*.

The response from those API will be in JSON format containing array of object. Each element of the array is correlated to table's row.

The table structure is presented at Listing 7:

Listing 7. Testcase Table Structure
```
TABLE: genres
FIELDS:
  - id
  - name

TABLE: books
FIELDS:
  - id
  - title
  - genreId
  - authorId

TABLE: author
FIELDS:
  - id
  - name
```

Using these API endpoints, we provide three problems. The first problem is named *g* problem. In order to solve *g* problem, we should transform HTTP API endpoint's response into array of genre's name (i.e: *["fiction","history","science"]*). In *g* problem, only one endpoint is involved.

The second problem is named *gb* problem. In *gb* problem we should fetch book titles for each genres and present it in the following manner: *["name":"fiction","books":["Rise of the Rebels","A New Dawn"],"name":"history","books":["John Adams","1776"],...]*. This problem is more difficult than the first problem, since we have to deal with two endpoints (genres and books).

The last problem named *gba* . It is very similar to *gb* problem, but in this case we should also show the author's name of every book. The expected result is as follow: *["name":"fiction","books":["title":"Rise of the Rebels","author":"Michael Kogge","title":"A New Dawn","author":"John Jackson Miller"],...]*

In order to measure the readability of the solutions, we use Flesch Kincaid readability test. while to measure the performance of the solutions, we use UNIX *time* utility.

B. Solutions

For each problems, we build 3 solutions. The solutions are using HTTP API protocol and written in three programming languages (Python, JavaScript, and CHIML). Some technologies like CORBA, BPEL, and EJB are not helpful to solve our problems since those technologies require IDL, broker setup, and other prerequisites. Like HTTP API, SOAP and RPCs are just protocols. Given the correct data encoder/decoder and suitable end point, we can also build the same solutions for these technologies.

In order to make the code shorter, we use *map* and *arrow function* for JavaScript, and using *lambda expression* for Python.

The solutions of problem *g* are presented in Listing 8, 9, and 10:

Listing 8. JavaScript Solution for problem-g
```
const rpn = require('request-promise-native')
async function fetchGenres () {
  const body = await rpn('http://localhost:3000/
      genres')
  const genres = JSON.parse(body).map((genre) =>
      genre.name)
```

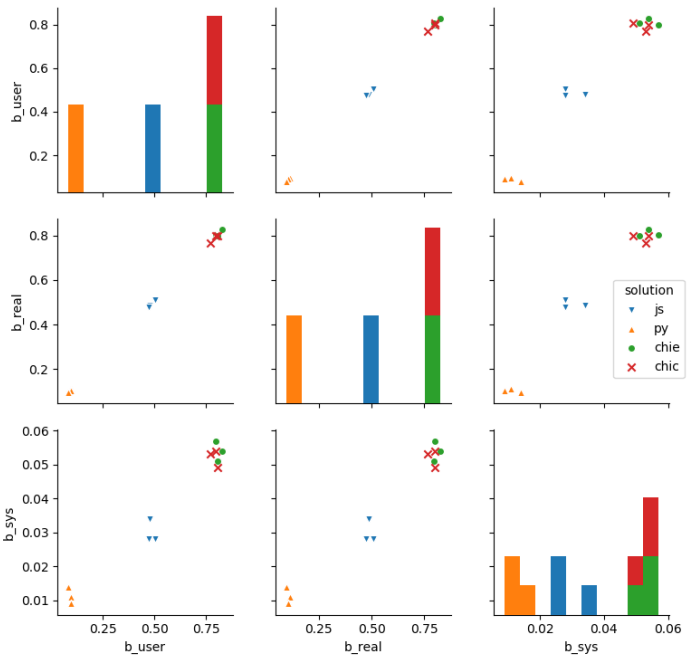

Fig. 1. Performance Comparison

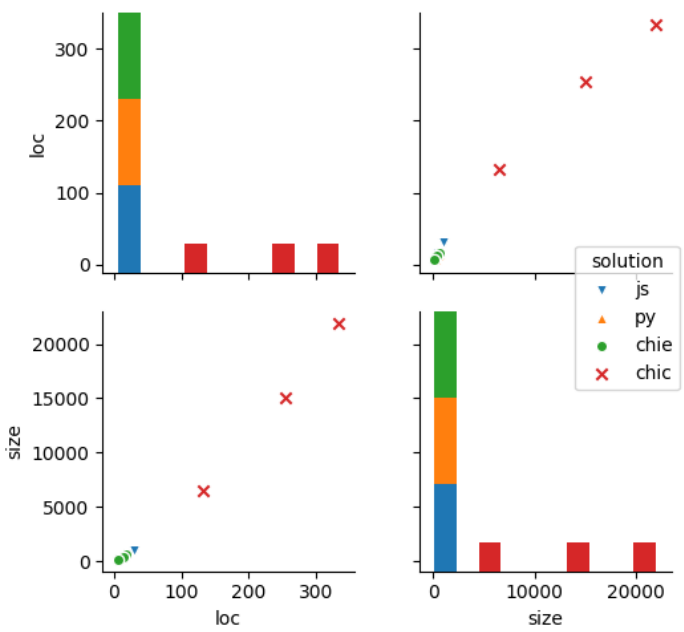

Fig. 2. Size Comparison

```
  console.log(JSON.stringify(genres))
}
fetchGenres()
```

Listing 9. Python Solution for problem-g

```
import json
from urllib import request
response = request.urlopen('http://localhost:3000/
    genres').read()
genres = json.loads(response)
result = list(map(lambda genre: genre['name'],
    genres))
print(json.dumps(result))
```

Listing 10. CHIML Solution for problem-g

```
out: genres
do:
  - ('http://localhost:3000/genres') -> [sys.
      httpRequest] -> genres
  - map: genres
    into: genres
    do: (genre) -> (x) => x.name -> genre
```

VI. Result and Discussion

After building the solutions, we compare the performance by using UNIX *time* command. This benchmark providing 3 numbers, *sys*, *user*, and *real*. *Sys* is the amount of CPU time spent within kernel. *User* is the amount of CPU time spent outside the kernel but within the process. And *real* is the clock time from start to finish the call. *Real* also include time spent for IO process. We named these performance *b_sys*, *b_real*, and *b_user* respectively.

Besides comparing the performance, we also calculate the file size in byte (*size*), number of lines in code (*loc*), Flesch reading ease (*fre*), Flesh-Kincaid grade level (*fkgl*), and Raymond Buse's measurement (*rbr*).

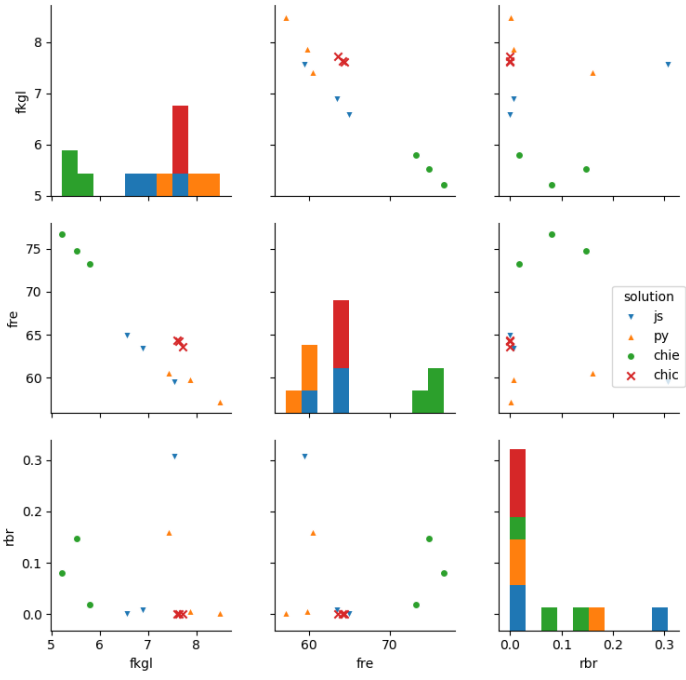

Fig. 3. Readability Comparison

TABLE I. COMPARISON

b_real	b_sys	b_user	fkgl	fre	loc	rbr	size	solution/case
0.509	0.028	0.503	6.571	64.934	30	0.000	978	js, gba
0.114	0.011	0.096	8.492	57.214	15	0.002	662	py, gba
0.827	0.054	0.829	5.795	73.210	17	0.018	576	chie, gba
0.799	0.054	0.798	7.606	64.371	334	0.000	21955	chic, gba
0.486	0.034	0.477	6.895	63.440	16	0.007	540	js, gb
0.104	0.009	0.092	7.871	59.853	12	0.006	471	py, gb
0.802	0.057	0.798	5.219	76.690	13	0.081	396	chie, gb
0.801	0.049	0.806	7.639	64.218	254	0.000	15048	chic, gb
0.476	0.028	0.474	7.551	59.499	7	0.307	250	js, g
0.096	0.014	0.081	7.422	60.549	6	0.160	217	py, g
0.798	0.051	0.807	5.518	74.796	6	0.147	167	chie, g
0.768	0.053	0.770	7.715	63.576	133	0.000	6498	chic, g

For *b_sys*, *b_real*, and *b_user*, smaller number means better performance. For *size* and *loc*, smaller number also means better. For *fre*, greater numbers mean that the program is more readable. However, for *fkgl*, smaller number mean smaller years of education required. This means that *fkgl* and *fre* should has negative correlation.

The result is presented in the table I. In the table, there are two CHIML solutions for each cases. For the first solution *chie* , we executed the solution directly, while for the second solution *chic* we transpiled it first into JavaScript.

As shown in figure 1, we can conclude that Python solution is unexpectedly outperforming Javascript and CHIML. This is interesting, since in our solutions we didn't use multithread strategy. Python was expected to be slower because it has no non-blocking mechanism by default. However, the experiment shows different result.

Also, *b_user* and *b_real* is strongly correlated to each other. This makes sense since the majority CPU time is spent in *user* mode.

In term of performance, CHIML solution is still the slowest. This also make sense, since under the hood, CHIML will be translated into JavaScript. However, we observe that for *chie* and *chic* , the execution speed are quite close to each other. Hence we conclude that the transpilation doesn't greatly improve the performance. It means that *vm.runInNewContext* is fast enough, and doesn't severely reduce the performance. Furtherly, we observe that the *module-resolve* mechanism take take almost 400 ms, thus severely reduces the execution-speed.

In terms of *size*, as shown in figure 2, CHIML outperforming Python and JavaScript. The same is also true for *fkgl* and *fre* (as shown in figure 3). Thus, we can conclude that according to Flesch-Kincaid readability test, CHIML is more readable from Python and JavaScript. As for *rbr* (Raymond Buse Readability Metrics), we see that CHIML has best values in *gba* case and wors values for *g* case. Thus, for a very simple cases, Python/Javascript could be a better alternative.

VII. CONCLUSION

CHIML serve well as orchestration language. However, for controlling structure, the existence of intermediary components can help to boost performance. The best trait of CHIML is it's support for programming-in-large and programming-in-small. Even tough the control structure is still suffering for speed and performance, it serves well as a prototyping tool. In term of flexibility, CHIML is comparable to Python or JavaScript. In most cases, it even has smallest size and LOC than both, Python and JavaScript. This mean that the developer can start orchestration solution in CHIML, and gradually do optimization as needed.

Despite its advantage, CHIML interpreter still has to be refactored and optimized in order to make it usable for real-world use-cases.

VIII. FUTURE WORK

One big challenge in software development is how to let developer transform a monolithic application into microservices or vice-versa. Monolithic system is a better choice for targetting small market, while, microservices-architecture is a better bet for targeting larger market. Developer can easily choose the best choice once they know the scope of the application. However, in most cases, developers need to consider that as business grow, the application also need to serve a larger market. In this case, the best solution is by building a monolithic application that can easily scaled up and refactored into microservices.

Thus, beside optimize the CHIML interpreter, we will also redesign Chimera-Framework into a platform that focus on how to make a smoother transition between monolithic and microservices-architecture.

REFERENCES

[1] D. H. Hansson, "Ruby on rails — a web-application framework that includes everything needed to create database-backed web applications according to the model-view-controller (mvc) pattern." https://rubyonrails.org, accessed: 2018-04-30.

[2] D. S. Foundation, "The web framework for perfectionists with deadlines — django," https://djangoproject.org, accessed: 2018-04-30.

[3] T. Otwell, "Laravel - the php framework for web artisans," https://laravel.com, accessed: 2018-04-30.

[4] B. K. Jr., "What is software vendor lock-in? (and how to avoid it)," https://www.makeuseof.com/tag/software-vendor-lock-avoid, accessed: 2018-04-30.

[5] M. Little, "Soa versus microservices?" https://www.infoq.com/news/2015/12/soa-v-microservices, accessed: 2018-04-30.

[6] J. Spacey, "Orchestration vs choreography," https://simplicable.com/new/orchestration-vs-choreography, accessed: 2018-04-30.

[7] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 1968, pp. 88–98.

[8] Trac, "Json-rpc," http://json-rpc.org, accessed: 2018-04-30.

[9] I. UserLand Software, "Xml-rpc.com," http://xmlrpc.scripting.com, accessed: 2018-04-30.

[10] I. Google, "grpc.io," https://grpc.io, accessed: 2018-04-30.

[11] T. Feilhauer and M. Sobotka, "Def-a programming language agnostic framework and execution environment for the parallel execution of library routines," *Journal of Cloud Computing*, vol. 5, no. 1, p. 20, 2016.

[12] G. F. Gunawan, J. F. Palandi, and M. Amien, "Chimera - simple language agnostic framework for stand alone and distributed computing," in *Computer Applications and Information Processing Technology (CAIPT), 2017 4th International Conference on*. IEEE, 2017, pp. 144–153.

[13] C. C. Evans, "The official yaml website," http://yaml.org/, accessed: 2018-04-30.

[14] F. DeRemer and H. Kron, "Programming-in-the large versus programming-in-the-small," *SIGPLAN Not.*, vol. 10, no. 6, pp. 114–121, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808431

[15] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 407–416. [Online]. Available: http://doi.acm.org/10.1145/337180.337228

[16] F. R. R. R. C. B. Kincaid, J.P., "Derivation of new readability formulas (automated readability index, fog count, and flesch reading ease formula) for navy enlisted personnel," *Research Branch Report 875*, 1975.

[17] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010.