Jurnal Elektronik Ilmu Komputer Udayana
Volume 12, No 3. Februari 2024

p-ISSN: 2301-5373
e-ISSN: 2654-5101

CI/CD Implementation On Microservices For Increasing Availability On Big Data Processing

Kompiang Gede Sukadharma[a1], I Putu Gede Hendra Suputra[a2]

[a]Informatics, Udayana University
Jl. Raya Kampus UNUD, Bukit Jimbaran, Kuta Selatan, Badung, Bali, Indonesia
[1]kompiang.sukadharma@gmail.com
[2] hendra.suputra@unud.ac.id

Abstract

Big Data Processing needed a reliable system that not let the data loss. But sometimes we need to make the system down for a while because we need to push the newest changes of the system. The automation will help us achieve that. Continuous Integration and Continuous Deployment help us to reduce the downtime and increase the availability of the system. Thus, the implication will be led to reduce of data loss. This research focusses on the implementation of CI/CD Pipeline on single-point-of-failure service on Microservices Architecture. This research use Load-Testing to measure data loss on certain amount of time. The result on this research show that implementing CI/CD Pipeline on the Microservices that we made, make the down time will be less than 45 Second with 20 Virtual user who send the data.

Keywords: *Microservices, CI/CD, Load Test, GitHub Actions, Single-Point-of-Failure*

1. Pendahuluan

Seiring dengan meningkatnya kebutuhan dari industri teknologi, hal tersebut juga membuat banyak perubahan untuk beradaptasi terhadap perubahan yang perlu dilakukan. Tingginya perubahan yang terjadi membuat waktu yang digunakan untuk mengirimkan perubahan terbaru ke pengguna menjadi hal yang krusial. Perubahan yang terjadi memiliki kemungkinan membuat sebuah layanan tidak dapat diakses. Dalam pemrosesan Big Data, diperlukan arsitektur yang andal dan tingkat *availability* yang tinggi agar data-data dari pengguna tidak hilang. Oleh karena itu, diperlukan metode yang tepat untuk mengirimkan perubahan ke pengguna agar tidak mengganggu layanan yang sedang berjalan. Dalam pemrosesan Big Data, kita tidak bisa melakukannya secara konvensional [1]. Hal tersebut didukung dengan beberapa variabel yang telah teridentifikasi, yaitu (1) *volume*, (2) *velocity*, (3) *variety*, (4) *veracity* [2]. Variabel tersebut selanjutnya disebut sebagai sifat-sifat yang dimiliki oleh Big Data.

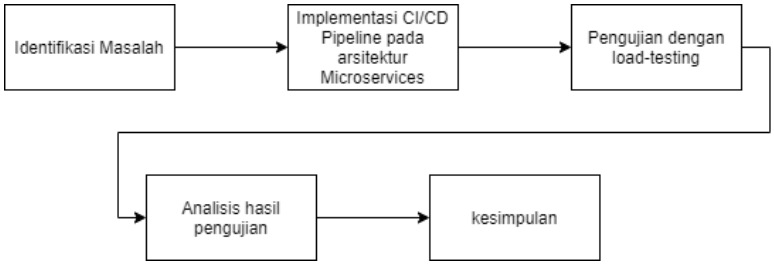
Tingginya tingkat *velocity* dan besarnya *volume* ketika melakukan pemrosesan Big Data menjadi tantangan untuk mengirimkan rilis terbaru ke pengguna. Untuk itu, praktik *continous* diharapkan menyediakan beberapa kelebihan, seperti: (1) Mendapatkan tanggapan lebih banyak dan lebih cepat dalam proses pengembangan dan juga dari pengguna, karena perubahan terkirim lebih cepat; (2) Merilis perubahan secara cepat dan reliabel yang mana akan meningkatkan kualitas produk; (3) *Continuous Deployment* atau CD akan mempercepat proses perubahan lingkungan aplikasi dari *development* dan *production* [3]; (4) Layanan yang sedang berjalan tidak mengalami *downtime* yang lama, jadi data yang sedang terkirim tidak banyak yang hilang.

Grab, salah satu *startup* yang memiliki layanan *superapp* yang bergerak di bidang transportasi, pengantaran makanan, dan *payment solutions*. Pada Agustus tahun 2022 mencatat 23.6 Juta transaksi pengguna [4]. Dengan banyaknya transaksi tersebut perusahaan ini menggunakan praktik *Continous* untuk mengirimkan perubahan rilis dengan lebih cepat agar operasi ini tidak memengaruhi kecepatan dan stabilitas mereka. Pada artikel yang dirilis oleh Grab, terdapat peningkatan banyaknya perubahan yang dirilis dari *staging environment* ke *production environment* sebanyak 14,6% dari hanya 10% perubahan yang diimplementasikan menggunakan metode konvensional pada tahun 2018 menjadi 24.6% perubahan yang diimplementasikan dari *staging* ke *production* menggunakan teknik CI/CD. Pada artikel disebutkan bahwa mereka menghemat 5000 hari kerja pda tahun 2020 sendiri. Hal tersebut setara dengan 20 tahun kerja [5].

2. Metodologi Penelitian

2.1 Alur Penelitian

Penulis mengikuti alur penelitian yang terdapat pada **Gambar 1** agar penelitian dapat berjalan secara lancar dan runtut



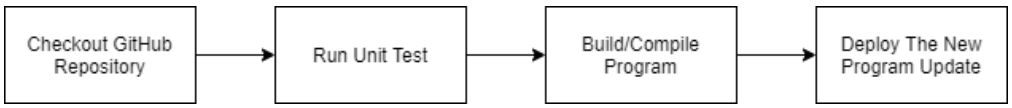
Gambar 1. Alur penelitian

Penjelasan dari alur penelitian di atas sebagai berikut:

a. Identifikasi Masalah
   Pada awal penelitian, penulis melakukan identifikasi masalah yang ada di lapangan dan juga melakukan riset dengan menggunakan literatur yang dapat membantu penelitian. Lalu, masalah yang ditemukan oleh penulis adalah besarnya kemungkinan *data loss* pada layanan *single point of failure* yang terjadi ketika suatu layanan dalam *microservices* dimatikan untuk mengirimkan perubahan yang terbaru.
b. Implementasi CI/CD Pipeline Pada Arsitektur Microservices
   Pada tahapan ini, penulis membuat sebuah sistem dengan arsitektur *microservices* dan juga *pipeline* CI/CD untuk menyimulasikan bagaimana CI/CD dapat meningkatkan *availibility* pada sebuah sistem. Implementasi menggunakan beberapa teknologi, untuk membuat sistem dengan arsitektur *microservices* penulis menggunakan bahasa pemrograman Go dan juga VS Code. Lalu, untuk membuat pipeline CI/CD, penulis menggunakan GitHub Actions dan juga menggunakan *compute service* dari AWS sebagai *instance* untuk menerapkan sistem ini.
c. Pengujian Dengan Load Testing
   Pengujian dilakukan dengan metode *Load-Testing* menggunakan K6 dan memperhatikan beberapa metriks.
d. Analisis Hasil pengujian
   Peneliti melakukan analisis terhadap beberapa metriks untuk menarik sebuah kesimpulan
e. Kesimpulan
   Pada tahap ini, penulis menarik kesimpulan dari beberapa metriks untuk menarik kesimpulkan dari hasil penelitian ini

2.2 CI/CD Pipeline

Pengujian pada metode ini berada pada lingkup proses pengintegrasian dan pengiriman perubahan terbaru pada sistem arsitektur *microservices*. Proses pengintegrasian dan pengiriman perubahan terbaru dapat dibuat ke dalam sebuah alur kerja yang kita konfigurasikan agar prosesnya berjalan secara otomatis serta dapat meminimalkan *downtime* dari sebuah service. Selain itu, pada penelitian ini penulis juga menggunakan metode ini untuk meneliti *data loss* yang diharapkan metode ini dapat meminimalkan hal tersebut. Alat yang digunakan dalam lingkup penelitian ini untuk menjalankan CI/CD Pipeline pada penelitian kali ini adalah GitHub Actions. **Gambar 2** menunjukkan alur kerja yang akan kita implementasikan menggunakan CI/CD Pipeline.



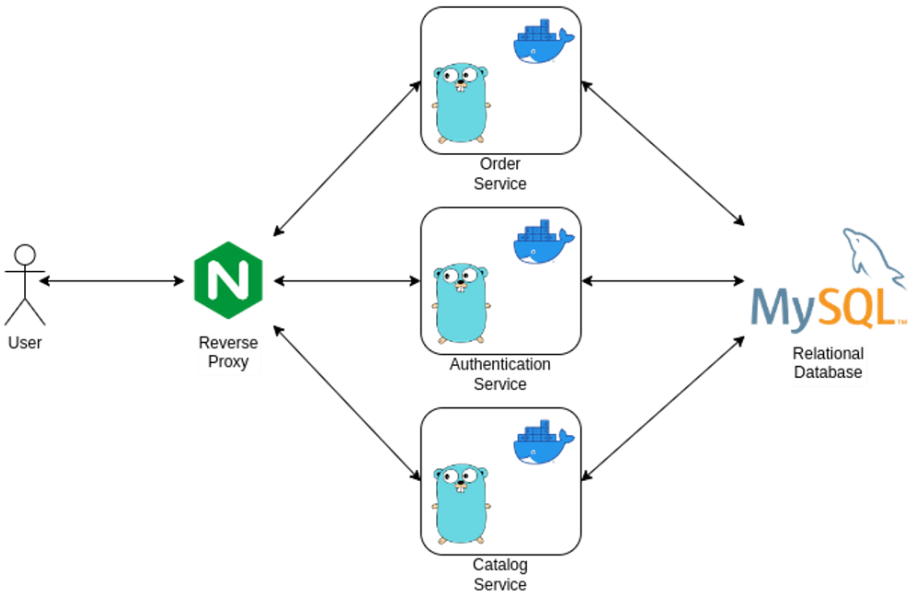
Gambar 2. Workflow CI/CD Pipeline

2.3 Arsitektur Microservices

Penelitian ini menggunakan arsitektur *microservices* untuk melihat dampak sebuah *service* yang memiliki sifat *single point of failure* atau sebuah *service* yang apabila terjadi kegagalan, kegagalan tersebut akan mengganggu kinerja dari sebuah sistem. Hal tersebut merupakan hal yang fatal pada arsitektur *microservices*. *Microservices* merupakan sebuah model arsitektur [6]. Ide utama dari

p-ISSN: 2301-5373
e-ISSN: 2654-5101

arsitektur jenis ini adalah memecah sistem yang lebih besar menjadi sistem-sistem yang lebih kecil. Hal ini dapat mengurangi kompleksitas dari sistem [3], membuat sistem yang lebih kecil menjadi lebih independen dan dapat membuat hubungan antar sistem atau *service* menjadi lebih renggang. Selain itu, sistem yang independen berimplikasi terhadap proses pengembangan dan *deploy* yang lebih mandiri. Lalu, kelebihan lain yang dimiliki oleh *microservices* adalah ketika sebuah *service* mengalami kegagalan, kegagalan tersebut hanya terisolasi dan tidak mengganggu *service lain*, kecuali *service* tersebut adalah *single point of failure.* **Gambar 3** merepresentasikan arsitektur yang penulis gunakan pada penelitian ini dengan mengimplementasikan *microservices* serta menjadikan Authentication Service menjadi sebuah *single point of failure.*



Gambar 3. Arsitektur *Microservices*

2.4 Docker

Penelitian ini menggunakan teknologi kontainerisasi untuk melakukan isolasi pada setiap *service* yang digunakan. Teknologi kontainerisasi memungkinkan kita untuk membuat sebuah virtualisasi pada tingkat aplikasi yang akan berjalan secara konsisten pada setiap infrastruktur. Kontainerisasi juga hanya akan membungkus aplikasi dengan aplikasi yang kita perlukan saja, hal tersebut akan membuat kontainer ini menjadi lebih ringan dan lebih cepat dimulai daripada virtualisasi pada tingkat sistem operasi. Salah satu *container engine* dan juga yang digunakan pada penelitian ini adalah Docker.

2.5 Load-Testing

Pengujian yang dilakukan pada penelitian ini menggunakan metode *Load-Testing* dengan bantuan K6. Metode *Load-Testing* akan menyimulasikan pengguna yang mengirim *request* sesuai dengan parameter yang diberikan ke program [7]. K6 merupakan alat bantuan untuk melakukan *Load-Testing* yang bersifat *open-source.* Dengan ini kita bisa mengetes performa dan keandalan sistem untuk mencari tahu masalah lebih awal. K6 akan menghasilkan kembalian berupa beberapa metris yang berkaitan tentang *request* yang telah dikirim. Metris ini akan membantu penulis untuk mengukur *data loss* yang terjadi pada penelitian ini. Beberapa metris yang digunakan pada penelitian dapat dilihat pada **Tabel 1**

Tabel 1. Metris Load-Testing

Key	Deskripsi
http_req_failed	Metris yang merepresentasikan banyaknya data yang berhasil dikirim dan banyaknya data yang gagal dikirim pada HTTP Request

3. Hasil dan Pembahasan

Berdasarkan alur penelitian yang dijelaskan pada bab sebelumnya. Maka hasil penelitian yang peneliti lakukan dapat dijabarkan sebagai beriku

3.1. Identifikasi Masalah

Pada penelitian ini penulis melakukan identifikasi masalah pada sistem yang menangani pemrosesan Big Data. Dengan mengidentifikasi sifat-sifat yang dimiliki oleh Big Data, seperti volume dan *velocity* yang dimiliki oleh Big Data serta juga mengidentifikasi perubahan yang terjadi pada industri yang sangat cepat. Penulis mengidentifikasi adanya masalah apabila terdapat perubahan yang diimplementasikan ketika sebuah layanan sedang melakukan pemrosesan data. Ketika mengimplementasikan perubahan pada sebuah layanan, layanan tersebut harus dimatikan terlebih dahulu. Pada layanan yang memproses Big Data, hal tersebut sangat fatal, karena ketika layanan dimatikan, data-data yang dikirimkan dari pengguna layanan akan hilang. Oleh karena itu, penulis memikirkan solusi untuk mempercepat proses pengimplementasian tersebut menggunakan CI/CD *pipeline*. CI/CD *pipeline*, akan membantu untuk meningkatkan *availibility time* dari sebuah layanan ketika dilakukan implementasi dari sebuah perubahan. Oleh karena itu, perubahan yang diimplementasikan akan lebih cepat sampai ke pengguna dengan waktu *down time* yang sedikit.

3.2. Implementasi CI/CD Pada Arsitektur Microservice

Sesuai dengan arsitektur *microservices* yang sudah penulis jelaskan pada bab sebelumnya, penulis akan melakukan implementasi perubahan terbaru terhadap Authentication Service. Authentication Service pada arsitektur di atas bersifat *single point of failure*. Oleh karena itu, Authentication Service perlu memiliki *down time* yang kecil agar layanan ini tidak mengganggu layanan lain bekerja. Untuk itu, penulis mengimplementasikan CI/CD *pipeline* menggunakan GitHub Actions. Intisari dari implementasi CI/CD *pipeline* dapat dilihat pada kode di **Tabel 2**.

Tabel 2. Implementasi CI/CD

	Kode
1	name: Deploy Auth Service
2	
3	on:
4	push:
5	tags:
6	- auth*
7	
8	env:
9	DOCKER_IMAGE_REPOSITORY_NAME: snatia-ci-cd
10	DOCKER_SERVICE_NAME: auth_service
11	
12	jobs:
13	build:
14	runs-on: ubuntu-latest
15	steps:
16	- name: Checkout Repository
17	uses: actions/checkout@v2
18	- name: Get Tag Name
19	run: echo "RELEASE_VERSION=${GITHUB_REF#refs/tags/}" >> $GITHUB_ENV
20	- name: Set up Go
21	uses: actions/setup-go@v2
22	with:
23	go-version: 1.18
24	- name: Run the unit test for this project
25	run: go test ./... -cover
26	- name: Login to Docker Hub
27	uses: docker/login-action@v1
28	with:
29	username: ${{ secrets.DOCKER_HUB_USERNAME }}
30	password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}
31	- name: Set Up Docker Buildx
32	uses: docker/setup-buildx-action@v1
33	- name: Build and push
34	uses: docker/build-push-action@v2
35	with:

p-ISSN: 2301-5373
e-ISSN: 2654-5101

```
36          context: .
37          file: ./auth.Dockerfile
38          push: true
39          tags: ${{ secrets.DOCKER_HUB_USERNAME }}/${{
env.DOCKER_IMAGE_REPOSITORY_NAME }}:${{ env.RELEASE_VERSION }},${{
secrets.DOCKER_HUB_USERNAME }}/${{ env.DOCKER_IMAGE_REPOSITORY_NAME
}}:auth-latest
40      - name: Executing remote command using ssh
41        uses: appleboy/ssh-action@master
42        with:
43          host: ${{ secrets.SSH_SERVER }}
44          username: ${{ secrets.SSH_USERNAME }}
45          password: ${{ secrets.SSH_PASSWORD }}
46          port: 22
47          script: |
48            cd deploy/
49            docker compose rm -sf ${{ env.DOCKER_SERVICE_NAME }}
50            docker rmi ${{ secrets.DOCKER_HUB_USERNAME }}/${{
env.DOCKER_IMAGE_REPOSITORY_NAME }}:auth-latest
51            docker compose up -d ${{ env.DOCKER_SERVICE_NAME }}
52  name: Deploy Auth Service
```

Implementasi dari CI/CD dapat di lihat pada **Tabel 2**. Penulis akan mengimplementasikan *pipeline* yang dijelaskan juga pada bab 2.2. Pada baris pertama, penulis memberikan nama terhadap *pipeline* ini, hanya sebuah metadata. Setelah itu, pada baris ke-3 hingga ke-6, penulis memberitahukan tentang *trigger* yang perlu didengarkan oleh GitHub Actions. Jadi, ketika *trigger* atau *event* tersebut terjadi, *workflows* ini akan berjalan. Selanjutnya, baris 16 hingga baris 17, penulis melakukan *checkout* terhadap *repository* dari GitHub yang terbaru. Baris 18 hingga 19, kita mengambil *tag* yang akan kita gunakan nantinya. Kemudian, baris ke-20 hingga baris ke-25 *hosted runner* mempersiapkan lingkungan untuk menjalan bahasa pemrograman Go, kemudian Go akan menjalankan semua test yang ada di *project* tersebut. Kemudian baris ke-26 hingga baris ke-39, penulis mempersiapkan lingkungan Docker untuk melakukan *compile program* dan mengirimkannya ke Docker Registry yang akan diambil oleh server nantinya. Lalu, baris ke-40, hingga barus ke-46 berguna untuk mempersiapkan koneksi SSH. Setelah itu, baris ke-47 hingga ke-52 akan melakukan *deployment* terhadap perubahan yang terbaru.

3.3. Tahap Load-Testing

Pengujian pada penelitian ini dilakukan dengan menggunakan *Load-Testing* menggunakan K6. K6 menggunakan JavaScript untuk menuliskan test yang akan digunakan. Penggalan kode dari *Load-Testing* menggunakan K6 sudah penulis lampirkan pada Tabel 3

Tabel 3. Penggalan Kode Load Testing

```
1  import http from 'k6/http';
2
3  export const options = {
4    vus: 20,
5    duration: '45s'
6  }
7
8  export default function () {
9    const base = {
10     baseUrl: "http://108.136.231.153:8000",
11     auth: "/auth",
12     order: "/order"
13   }
14
15   const payload = {
16     authPayload: {
```

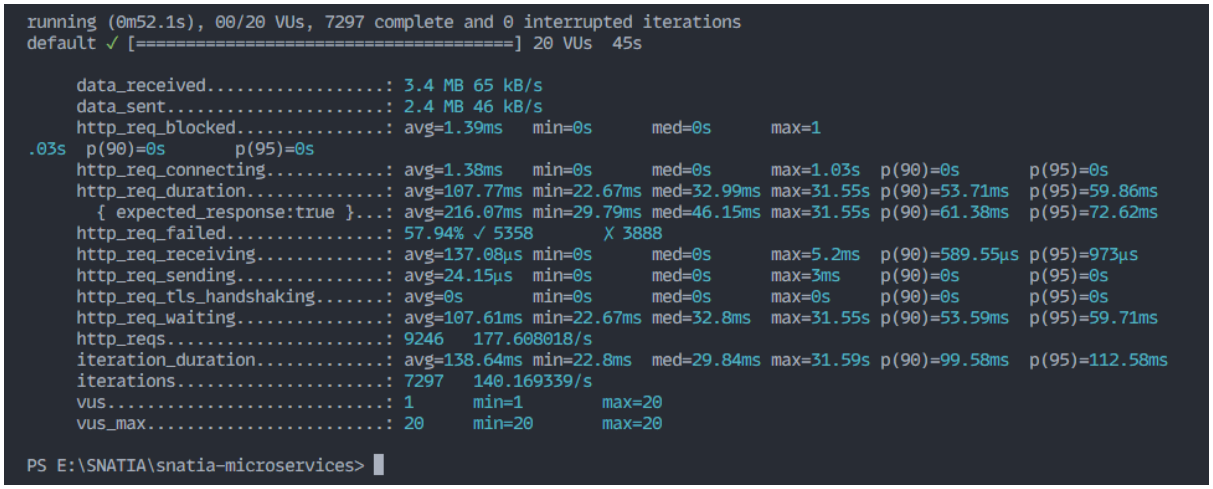
```
      "username": "kompiangg",
      "password": "12345678"
    },
    orderPayload: {
      "order": [
        {
          "item_id": 1,
          "price": 100000,
          "quantity": 2,
          "total_price": 200000
        }
      ]
    }
  }

  const res = http.post(
      base.baseUrl + base.auth + '/login',
      JSON.stringify(payload.authPayload), {
      headers: {'Content-Type': 'application/json'}
    }
  );

  const accessToken = res.json().data.access_token;
  http.post(
      base.baseUrl + base.order + '/orders',
      JSON.stringify(payload.orderPayload), {
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + accessToken
      }
    }
  );
}
```

Penggalan kode di atas merupakan *Load-Test* yang penulis gunakan. Pada baris ke-3 hingga ke-7 dijelaskan parameter yang akan digunakan untuk melakukan *Load-Test*, yaitu ‘vus’ yang berarti banyaknya user yang akan mengirimkan paket request. Lalu, parameter ‘durations’, yaitu lamanya user akan mengirimkan paket data. Penulis menggunakan 45 detik, karena dari beberapa pengujian, proses yang dihabiskan untuk menjalakan pipeline, yaitu 1 menit dan proses deploy sendiri menghabiskan waktu 30 detik, karena keterbatasan *resource* yang dimiliki penulis, maka durasi yang digunakan hanya 45 detik. Lalu, di dalam default function terdapat dua buah *service* yang di-test, yaitu Login dan Order yang berada pada service yang berbeda tapi Order *service* memiliki ketergantungan terhadap *service Login*.

p-ISSN: 2301-5373
e-ISSN: 2654-5101



Gambar 4. Hasil Load-Testing CI/CD Pipeline

3.4. Analisis Hasil Test

Dalam pengukuran *availibility* ketika diimplementasikan CI/CD Pipeline pada penelitian ini, dapat kita buatkan sebuah tabel perbandingan data yang berhasil terkirim dan kembali (*response*) dan yang gagal kembali dalam waktu 45 Detik

Tabel 4. Test Result

Key	Berhasil	Gagal	Durasi	Virtual User
http_req_failed	3888	5358	45 Detik	20

4. Kesimpulan

Berdasarkan penelitian dan hasil analisis dari penelitian, dapat dilihat pada tabel 4, bahwa ketika CI/CD Pipeline berjalan ketika masih terdapat *data stream* yang mengakses server, dalam 45 detik dengan 20 user didapatkan bahwa paket yang masih berhasil kembali sebanyak 3.888 dari 9.246 atau sebesar 42%. Hal ini menunjukkan bahwa CI/CD Pipeline dapat melakukan *deployment* pada sistem yang dibuat untuk penelitian ini kurang dari 45 detik. Lalu, CI/CD Pipeline dapat memperkecil waktu *downtime* ketika melakukan deployment daripada metode manual.

Referensi

[1] N. Shakhovska, N. Boyko, Y. Zasoba, and E. Benova, “Big data processing technologies in distributed information systems,” *Procedia Comput. Sci.*, vol. 160, no. 2018, pp. 561–566, 2019, doi: 10.1016/j.procs.2019.11.047.

[2] P. Muthulakshmi and S. Udhayapriya, “a Survey on Big Data Issues and Challenges,” *Int. J. Comput. Sci. Eng.*, vol. 6, no. 6, pp. 1238–1244, 2018, doi: 10.26438/ijcse/v6i6.12381244.

[3] M. Shahin, M. Ali Babar, and L. Zhu, “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,” *IEEE Access*, vol. 5, no. March, pp. 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.

[4] DMR, “Grab Statistics and Facts (2022),” 2022. https://expandedramblings.com/index.php/grab-facts-statistics/.

[5] S. Bougerel, “Our Journey to Continuous Delivery at Grab (Part 2),” *engineering.grab.com*, 2021. https://engineering.grab.com/our-journey-to-continuous-delivery-at-grab-part2.

[6] F. Rademacher, S. Sachweh, and A. Zundorf, “Differences between model-driven development of service-oriented and microservice architecture,” *Proc. - 2017 IEEE Int. Conf. Softw. Archit. Work. ICSAW 2017 Side Track Proc.*, pp. 38–45, 2017, doi: 10.1109/ICSAW.2017.32.

[7] H. Schulz, T. Angerstein, and A. Van Hoorn, “Towards automating representative load testing in continuous software engineering,” *ICPE 2018 - Companion 2018 ACM/SPEC Int. Conf. Perform. Eng.*, vol. 2018-January, pp. 123–126, 2018, doi: 10.1145/3185768.3186288.

This page is intentionally left blank.