

IMPLEMENTASI *MESSAGE BROKER* UNTUK MENDUKUNG *HIGH AVAILABILITY* PADA SISTEM DETEKSI PEMALSUAN DOKUMEN DI LINGKUNGAN DEPARTEMEN TEKNIK ELEKTRO UNIVERSITAS NEGERI MALANG

SKRIPSI

OLEH

MUHAMMAD SYUKUR ABADI

NIM 190535646043



**UNIVERSITAS NEGERI MALANG
FAKULTAS TEKNIK
DEPARTEMEN TEKNIK ELEKTRO DAN INFORMATIKA
PROGRAM STUDI TEKNIK INFORMATIKA
FEBRUARI 2023**

LEMBAR PERSETUJUAN SKRIPSI

Skripsi oleh Muhammad Syukur Abadi ini telah diperiksa dan disetujui untuk diujikan.

Malang, 1 Desember 2022

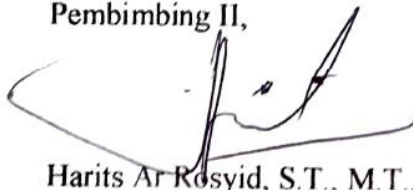
Pembimbing I,



Utomo Pujianto, S.Kom., M.Kom.

Malang, 1 Desember 2022

Pembimbing II,



Harits Ar-Rosyid, S.T., M.T., Ph.D

LEMBAR PENGESAHAN SKRIPSI

Skripsi oleh Muhammad Syukur Abadi ini telah dipertahankan di depan dewan penguji pada tanggal 22 Februari 2023.

Dewan Penguji

Muhammad Jauharul Fuady, S.T., M.T., Ketua
NIP. 198305072009121002

Utomo Pujiyanto, S.Kom., M.Kom., Anggota I
NIP. 198206042012121001

Harits Ar Rosyid, S.T., M.T., Ph.D., Anggota II
NIP. 198108112009121003

Mengesahkan,

Dekan Fakultas Teknik

Mengetahui

Ketua Departemen Teknik Elektro Dan
Informatika

Prof. Dr. Andoko, S.T., M.T.

NIP. 196508121991031005

Dr. Ir. Triyanna Widiyaningtyas, S.T., M.T.

NIP. 197412152008122002

PERNYATAAN KEASLIAN TULISAN

Saya yang bertanda tangan di bawah ini:

Nama : Muhammad Syukur Abadi

NIM : 190535645043

Jurusan/Program Studi : Teknik Elektro dan Informatika/Teknik Informatika

Fakultas/Program : Teknik/S1

menyatakan dengan sesungguhnya bahwa skripsi yang saya tulis ini benar-benar tulisan saya, dan bukan merupakan plagiasi/falsifikasi/fabrikasi baik sebagian atau seluruhnya.

Apabila di kemudian hari terbukti atau dapat dibuktikan bahwa skripsi ini adalah hasil plagiasi/falsifikasi/fabrikasi baik sebagian atau seluruhnya, maka saya bersedia menerima sanksi atas perbuatan tersebut sesuai dengan ketentuan yang berlaku.

Malang, 15 Februari 2023

Yang membuat pernyataan

Muhammad Syukur Abadi

RINGKASAN

Abadi, Muhammad Syukur. 2022. Implementasi *Message Broker* Untuk Mendukung *High Availability* Pada Sistem Deteksi Pemalsuan Dokumen Di Lingkungan Departemen Teknik Elektro Universitas Negeri Malang

Kata Kunci: *Message Queue, High Availability, Microservice*

Plagiarisme merupakan tindakan penggunaan pemikiran atau produk dari pihak lain tanpa melakukan atribusi terhadap pencipta produk tersebut. Pertumbuhan tersebut juga beriringan dengan jumlah data yang diproses, sehingga mendorong kebutuhan untuk membangun sistem yang mampu menangani jumlah data tersebut. Penelitian ini bertujuan untuk membangun dan menguji aplikasi berbasis *microservice* menggunakan *message queue*. Penelitian ini juga menggunakan *container* untuk mendukung *high availability*. Proses pengujian sistem pada penelitian ini menggunakan metode *load testing* dengan membandingkan implementasi *message queue* dan *REST* pada sistem yang dibangun. *Load testing* pada penelitian menggunakan bantuan aplikasi *Apache JMeter*. Hasil dari penelitian ini menunjukkan bahwa sistem yang dikembangkan dapat memisahkan ketergantungan dari proses *input* ke sistem deteksi plagiasi, serta hasil performa *message queue* menunjukkan *RabbitMQ* memiliki konsumsi memori yang stabil dan *Kafka* memiliki *latency* yang rendah.

SUMMARY

Abadi, Muhammad Syukur. 2022. Implementation Of Message Broker To Provide High Availability On Plagiarism Detection System At State University Of Malang.

Kata Kunci: *Message Queue, High Availability, Microservice*

Plagiarism is an act of using ideas or products from other parties without attribution to the creator of the product. This growth is also in tandem with the amount of data processed, thus driving the need to build a system capable of handling this amount of data. This study aims to build and test microservice-based applications using message queues. This research also uses containers to support high availability. The process of testing the system in this study uses the load testing method by comparing the implementation of message queues and REST on the system being built. Load testing in research uses the help of the Apache JMeter application. The results of this study indicate that the developed system can separate the dependence of the input process on the plagiarism detection system, and the message queue performance results show that RabbitMQ has stable memory consumption and Kafka has lowest latency.

KATA PENGANTAR

Puji serta syukur penulis panjatkan kepada Allah atas limpahan rahmat, izin, dan karunia-Nya, penulis dapat menyelesaikan skripsi dengan judul “Implementasi *Message Broker* Untuk Mendukung *High Availability* Pada Sistem Deteksi Pemalsuan Dokumen Di Lingkungan Departemen Teknik Elektro Universitas Negeri Malang”. Tidak lupa, penulis mengucapkan terima kasih kepada seluruh pihak yang telah mendukung penulis dalam menyelesaikan skripsi ini, khususnya kepada:

1. Ibu Dr. Ir. Triyanna Widiyaningtyas, S.T., M.T, selaku Ketua Departemen Teknik Elektro Dan Informatika Universitas Negeri Malang.
2. Bapak Dr. Eng. Didik Dwi Prasetya, S.T., M.T, selaku Koordinator Program Studi S1 Teknik Informatika Universitas Negeri Malang.
3. Bapak Utomo Pujiyanto, S.Kom., M.Kom., selaku Dosen Pembimbing 1 yang telah memberikan arahan, bimbingan, motivasi, dan masukan selama proses penyusunan skripsi ini.
4. Bapak Harits Ar Rosyid, S.T., M.T., Ph.D., selaku Dosen Pembimbing 2 yang telah memberikan arahan, bimbingan, motivasi, dan masukan selama proses penyusunan skripsi ini.
5. Ibu Kartika Candra Kirana, S.Pd., M.Kom. selaku dosen pembimbing akademik yang telah memberikan motivasi dan bimbingan sejak semester 1.
6. Kedua orang tua penulis yang selalu memberikan dukungan.
7. Rekan seperjuangan tim penelitian *pre-plagiarism checker* yaitu Saddam dan Jaka serta *circle PDF* Perjuangan yang berjuang bersama di bawah payung “Rapatkan Paragraf *Cumlaude* Di Kepalan Tangan”.
8. Bapak dan Ibu Dosen Departemen Teknik Elektro Dan Informatika Universitas Negeri Malang yang telah memberi ilmu dan membimbing penulis selama penulis menempuh pendidikan di Universitas Negeri Malang.

9. Teman-teman S1 Teknik Informatika Offering B Angkatan 2019 yang telah kebersamai penulis dalam menuntut ilmu.
10. Yolla Fitri Elmi, Fakhira Alanna Shabira, Fatiruna An-Fasya Jalil, dan Rizki Aqil Muhaimin, teman-teman Heptagon yang telah berjuang bersama-sama untuk menggapai cita-cita.
11. Muhammad Farhan Devasa, Satria Wahyu Prayoga, Dony Aristo Hamid Siregar, teman-teman Kolong Langit yang selalu menyempatkan hadir di waktu senggang penulis.
12. *Sunrise Studio* selaku studio yang merilis *Gundam* yang telah memberikan hiburan bagi penulis.
13. Diri sendiri yang tidak berhenti menunda kekalahan.

Dalam penyusunan skripsi ini, penulis menyadari bahwa masih banyak kesalahan dan kekurangan baik dari segi materi ataupun penulisan. Oleh karena itu, kritik yang bersifat konstruktif sangat dibutuhkan penulis untuk memperbaiki skripsi ini dan tulisan-tulisan di masa depan. Besar harapan penulis bahwa skripsi ini bisa bermanfaat tidak hanya untuk kalangan akademisi, tetapi juga orang banyak.

Malang, 17 Desember 2022

Muhammad Syukur Abadi

DAFTAR ISI

LEMBAR PERSETUJUAN SKRIPSI	ii
LEMBAR PENGESAHAN SKRIPSI	iii
PERNYATAAN KEASLIAN TULISAN	iv
RINGKASAN	v
SUMMARY	vi
KATA PENGANTAR	vii
DAFTAR ISI	ix
DAFTAR GAMBAR	xi
DAFTAR TABEL	xii
DAFTAR LAMPIRAN	xiii
BAB 1 PENDAHULUAN	1
1.1 Latar Belakang	1
1.2 Rumusan Masalah	3
1.3 Tujuan Penelitian.....	4
1.4 Batasan Masalah	4
1.5 Manfaat Penelitian.....	4
BAB 2 KAJIAN PUSTAKA	4
2.1. <i>Software Development Life Cycle</i>	4
2.2. <i>Microservice</i>	4
2.3. <i>High Availability</i>	5
2.4. <i>Message Queue</i>	7
2.5. <i>Container</i>	8
2.6. <i>NoSQL</i>	10
2.7. <i>Performance Testing</i>	12
BAB 3 METODOLOGI PENELITIAN	12
3.1. Analisis Kebutuhan Sistem	12
3.2. Desain Sistem	14
3.2.1 Arsitektur Sistem.....	14
3.2.2 Skema Basis Data.....	15
3.2.3 Logika Bisnis	16
3.3. Implementasi	17
3.4. Pengujian	17
3.4.1 Metrik Evaluasi	17

3.4.2	Skenario Pengujian.....	18
3.5.	Pemeliharaan	19
BAB 4	HASIL DAN PEMBAHASAN.....	18
4.1.	Hasil Implementasi Sistem.....	18
4.1.1.	Entry Point Service	18
4.1.2.	File Docker Compose.....	19
4.1.3.	Instrumentasi	19
4.2.	Hasil Pengujian Sistem.....	20
4.2.1.	Single Thread, Multiple Loop	21
4.2.2.	Single Loop, Multiple Thread	22
4.2.3.	Multiple Thread, Multiple Loop	23
4.3.	Analisis Implementasi & Hasil Pengujian Sistem.....	23
BAB 5	KESIMPULAN	26
5.1.	Kesimpulan.....	26
5.2.	Saran	26
DAFTAR PUSTAKA.....		27
LAMPIRAN		31

DAFTAR GAMBAR

Gambar 1 Diagram Alir Penelitian	12
Gambar 2 Desain Sistem.....	14
Gambar 3 Sequence Diagram Sistem	16

DAFTAR TABEL

Tabel 1 Message queue yang digunakan di kalangan industri.....	8
Tabel 2 Perbedaan Virtual Machine dan Container.....	9
Tabel 3 Jenis-Jenis NoSQL.....	10
Tabel 4 Spesifikasi Message Queue Yang Digunakan	13
Tabel 5 Skema Dokumen.....	15
Tabel 6 Perangkat Lunak Yang Digunakan Untuk Pengembangan	17
Tabel 7 Parameter Pengujian	18
Tabel 8 Perangkat Keras Yang Digunakan Untuk Pengujian.....	18
Tabel 9 API Contract	18
Tabel 10 Daftar Docker Image Yang Digunakan	19
Tabel 11 Metrik Prometheus Yang Digunakan	20
Tabel 12 Hasil Pengujian Skenario Single Thread, Multiple Loop.....	21
Tabel 13 Pengujian Skenario Single Loop, Multiple Thread	22
Tabel 14 Hasil Pengujian Skenario Multiple Loop Multiple Thread	23

DAFTAR LAMPIRAN

Lampiran 1. Hasil Pengujian Skenario Single Thread, Multiple Loop.....	31
Lampiran 2. Hasil Pengujian Skenario Single Loop, Multiple Thread.....	33
Lampiran 3. Hasil Pengujian Skenario Multiple Thread, Multiple Loop	35

BAB 1

PENDAHULUAN

1.1 Latar Belakang

Plagiarisme merupakan tindakan penggunaan pemikiran atau produk dari pihak lain tanpa melakukan atribusi dari mana pemikiran atau produk tersebut untuk memperoleh keuntungan yang tidak perlu diuangkan (Fishman, 2009). Perkembangan teknologi mendorong terjadinya tindakan plagiasi. Beberapa penelitian dilakukan untuk mencegah tindakan plagiasi, seperti kursus dan desain asesmen, penggunaan perangkat lunak untuk mencocokkan kata, serta pelatihan antiplagiasi dan peningkatan literasi akademik (Mahmud et al., 2019).

Perkembangan jumlah pengguna internet turut mendorong bertambahnya jumlah data yang tersimpan serta kemampuan komputasi untuk memproses data tersebut, namun sistem jaringan antaraplikasi tradisional tidak mampu memproses penambahan jumlah *traffic* tersebut (Moon & Shine, 2020). Semakin bertambah jumlah data dan kebutuhan komputasi, maka proses komputasi untuk memproses data tersebut akan menjadi terdistribusi (Gholamian & Ward, 2021).

Pada dasarnya, ada dua arsitektur perangkat lunak, yaitu *monolith* dan *microservice*. Aplikasi dengan arsitektur *monolith* menggunakan satu basis kode untuk keseluruhan aplikasi, sedangkan aplikasi dengan arsitektur *microservice* memecah aplikasi ke dalam aplikasi-aplikasi atau *service-service* yang lebih kecil (Blinowski et al., 2022).

Pola komunikasi yang digunakan pada arsitektur berbasis *microservice* dibagi menjadi sinkron dan asinkron. Pada pola komunikasi sinkron, *client* akan menunggu *server* selesai mengolah *request*, sedangkan pola komunikasi asinkron *client* tidak perlu menunggu *server* selesai mengolah *request*. Beberapa penelitian dilakukan untuk membandingkan performa pola

komunikasi sinkron dengan asinkron pada aplikasi *microservice*, seperti penelitian (Shafabakhsh et al., 2020) yang menunjukkan bahwa *throughput* aplikasi *microservice* yang menggunakan pola komunikasi sinkron lebih rendah dibandingkan dengan pola komunikasi sinkron.

Ada beberapa cara untuk membangun aplikasi berbasis *microservice*. Penelitian (Márquez & Astudillo, 2019), mengungkapkan ada 3 pola arsitektur sistem untuk membangun *microservice* yang menunjang *high availability*, diantaranya *circuit breaker*, *service registry*, dan *messaging*.

Message broker merupakan cara suatu aplikasi berkomunikasi dengan aplikasi secara asinkron (Blas, 2019). Hal ini memungkinkan aplikasi tetap dapat bertukar pesan walau salah satu aplikasi tidak dapat dijangkau, sehingga memungkinkan setiap aplikasi dapat berjalan secara independen, sehingga memberikan skalabilitas dan redundansi yang tinggi (Krzemien et al., 2019). Paradigma *message queue* yang digunakan dalam *message broker* dapat diaplikasikan sebagai penghubung antarproses komunikasi, sebagai cara untuk berkomunikasi antar-*service* dalam sistem komputasi terdistribusi. Selain itu, *message broker* dapat digunakan untuk memproses *big data* (Wang et al., 2020), *IoT*, serta aplikasi dengan arsitektur *event-driven* (John & Liu, 2017).

Message broker telah digunakan oleh beberapa pelaku industri untuk berbagai kepentingan, sebagai contoh LinkedIn memanfaatkan *message broker* untuk keperluan *data streaming* dan analisis data secara *offline*, Yahoo! Untuk keperluan analisis data secara *real time*, dan Netflix yang menggunakan *message broker* sebagai saluran untuk pengumpulan data (Hiraman et al., 2018). Sementara itu, pelaku industri di Indonesia seperti Gojek menggunakan *message broker* untuk mendeteksi *GPS* palsu.

Sistem deteksi pemalsuan dokumen bekerja dengan cara mencari karakter yang warna hurufnya sengaja dicetak putih menggunakan bantuan *library PyMuPDF* dan mencari karakter yang ditukar dengan *embedded font* menggunakan *OCR*. Sistem ini menerima *input* berupa *file PDF* dan

menghasilkan koordinat/*bounding box*, di mana koordinat tersebut menunjukkan lokasi pemalsuan dokumen. Sistem deteksi ini memakan waktu paling lama 2 menit untuk memeriksa keseluruhan dokumen. Sistem ini juga diharapkan mampu menangani *load* sebanyak jumlah pengguna aplikasi pendeteksi plagiasi seperti *Grammarly* sebanyak 30 juta pengguna¹ dan *Turnitin* sebanyak 34 juta pengguna².

Oleh karena Sistem Deteksi Pemalsuan Dokumen membutuhkan waktu untuk memeriksa dokumen serta kebutuhan untuk melayani jumlah *load* yang banyak, maka penelitian ini dilakukan untuk membangun sistem *backend* yang mampu menangani *load* dalam jumlah besar secara asinkron dan mendukung *high availability*. Salah satu cara untuk mencapai *high availability* pada aplikasi adalah dengan menggunakan *message broker* seperti yang dilakukan oleh (Sasidharan, 2022). Penelitian ini bertujuan untuk membangun sistem *backend* yang mengintegrasikan proses *input* dengan Sistem Deteksi Pemalsuan Dokumen menggunakan pola komunikasi asinkron dengan bantuan *message broker* dan membandingkan dengan pola komunikasi sinkron menggunakan *REST API*. Penelitian ini mengukur *throughput*, *latency*, serta konsumsi memori yang digunakan oleh *message broker* dan *REST server*.

1.2 Rumusan Masalah

Sebagaimana latar belakang yang telah dijabarkan, maka rumusan masalah pada penelitian ini sebagai berikut :

1. Mengembangkan sistem *backend pre-plagiarism checker* berbasis *container*.
2. Mengintegrasikan serta mengevaluasi sistem *backend pre-plagiarism checker* dengan *message queue* untuk mendukung *high availability*.

¹ <https://www.grammarly.com/about>

² <https://www.turnitin.com/press/advance-to-acquire-turnitin>

1.3 Tujuan Penelitian

Tujuan dari penelitian ini adalah membangun dan mengevaluasi sistem *backend* yang mampu mengintegrasikan proses *input user* dengan *pre-plagiarism checker engine* menggunakan *message queue* berbasis *container* untuk mendukung *high availability*.

1.4 Batasan Masalah

1. Aplikasi yang dibangun merupakan aplikasi *backend* dengan arsitektur *microservice* dan menggunakan teknologi *container* yang berjalan di *local*.
2. *Container-container* yang digunakan dalam tahap pengembangan adalah *single instance container*.
3. Protokol yang digunakan untuk komunikasi antara *client* dengan aplikasi *backend* yang dikembangkan adalah *HTTP*.

1.5 Manfaat Penelitian

Penelitian ini diharapkan dapat menjadi referensi dalam membangun aplikasi berbasis *container* dengan arsitektur perangkat lunak berbasis *microservice*, digunakan oleh institusi untuk mengintegrasikan proses *input* dari pengguna serta *output* yang dihasilkan sistem praplagiasi, serta memberikan gambaran luas mengenai implementasi *high availability* menggunakan arsitektur *messaging*.

BAB 2

KAJIAN PUSTAKA

2.1. *Software Development Life Cycle*

Proses pengembangan perangkat lunak membutuhkan metodologi guna merencanakan, mengatur, dan mengontrol proses pengembangan perangkat lunak. Metodologi ini dikenal dengan *software development life cycle* yang selanjutnya disebut *SDLC*. Ada beberapa model *SDLC* yang digunakan saat ini, seperti: *waterfall*, *RAP*, *RAD*, *agile*, dan *rapid* yang setiap modelnya memiliki kelebihan dan kekurangan. Walau begitu, *SDLC* memiliki kesamaan dalam prosesnya. Kesamaan tersebut adalah rangkaian langkah yang harus dijalankan oleh pengembang dalam rangka menghasilkan dan merilis perangkat lunak (ind, Karambir, 2015).

2.2. *Microservice*

Menurut hukum Conway, desain sebuah sistem dalam suatu organisasi mencerminkan cara komunikasi pada organisasi tersebut. Organisasi yang terikat berorientasi terhadap satu basis kode, sedangkan organisasi yang tidak terikat menggunakan basis kode yang terdistribusi (Maccormack et al., 2007).

Pada dasarnya, ada dua arsitektur perangkat lunak yang digunakan dalam pengembangan perangkat lunak, yaitu *monolith* dan *microservice*. Arsitektur *monolith* menggabungkan seluruh logika bisnis dalam satu basis kode, sedangkan arsitektur *microservice* memecah aplikasi menjadi beberapa bagian. Masing-masing *service* pada aplikasi *microservice* berjalan secara independen, memiliki logika bisnis sendiri, bahkan memiliki *database* sendiri (Blinowski et al., 2022).

Pola komunikasi yang digunakan pada aplikasi *microservice* dibagi menjadi komunikasi secara sinkron dan asinkron. Pola komunikasi secara sinkron dengan model *request/response* menggunakan *REST* atau *gRPC* (Bolanowski et al., n.d.). Sedangkan pola komunikasi yang terjadi secara

asinkron menggunakan *message queue*, di mana *client* tidak perlu menunggu *server* untuk selesai mengolah *request* (Karabey Aksakalli et al., 2021).

Dinamika yang terjadi dalam perangkat lunak berbasis *microservice* mendorong implementasi *monitoring*/pemantauan terhadap aktivitas sistem. *Monitoring* dapat dilakukan untuk memantau metrik seperti jumlah *request* yang berhasil dan gagal diproses, konsumsi sumber daya seperti memori dan *CPU*, atau jumlah koneksi ke *database* (Waseem et al., 2021).

Penelitian-penelitian terdahulu dilakukan untuk mengetahui performa aplikasi *monolith* dibandingkan dengan aplikasi *microservice*. Penelitian yang dilakukan oleh (Tapia et al., 2020) menggunakan teknologi *container* untuk menjalankan aplikasi dan pengujian aplikasi didasarkan pada performa aplikasi. Hasil penelitian tersebut menunjukkan bahwa aplikasi *microservice* mampu menangani beban *request* lebih cepat dibanding aplikasi *monolith*. Dalam segi penggunaan sumber daya seperti penggunaan *CPU*, memori, kecepatan transfer data, serta kecepatan membaca dan menulis di disk, aplikasi *microservice* lebih unggul dibandingkan dengan aplikasi *monolith*.

Aplikasi berbasis *monolith* memiliki kelemahan, salah satunya jika terdapat bagian program yang bermasalah dapat berdampak pada keseluruhan aplikasi, kom. Oleh karena itu, beberapa perusahaan berusaha untuk membagi aplikasi kedalam aplikasi-aplikasi atau *service-service* kecil yang membentuk aplikasi yang kompleks (Karabey Aksakalli et al., 2021).

2.3. High Availability

Availability merupakan durasi yang dapat ditoleransi saat terjadi masalah pada aplikasi. *Availability* ditinjau dari beberapa aspek, diantaranya waktu yang dibutuhkan untuk mengakses aplikasi, waktu yang dibutuhkan untuk mendapat respon yang valid, waktu yang

dibutuhkan untuk menerima data, jaminan penyimpanan dan pemeliharaan data secara terintegrasi, dan kemampuan aplikasi untuk melakukan *scaling* saat terjadi *traffic* yang tinggi (Berenberg & Calder, 2023).

Menurut (Márquez & Astudillo, 2019), ada beberapa pola arsitektur sistem yang dapat digunakan untuk menunjang *availability* dari perangkat lunak berbasis *microservice*, diantaranya :

1. *Circuit Breaker*: Pola ini memeriksa kesehatan suatu aplikasi dengan cara memeriksa jumlah *request* yang gagal diproses oleh perangkat lunak. Jika jumlah *request* yang gagal menyentuh ambang batas, maka perangkat lunak akan mengirim pesan *error* ke perangkat lunak yang mengirimkan *request* dan mematikan perangkat lunak yang memproses *request*.
2. *Service Registry*: Pola *service registry* memetakan pengenalan unik dari suatu *service* dengan *service* tersebut untuk memisahkan dependensi antara alamat fisik *service* dengan pengenalan unik *service*.
3. *Messaging*: Pola *messaging* menggunakan *message queue* untuk berkomunikasi antar-*service* secara asinkronus

Di samping penggunaan pola arsitektur sistem untuk meningkatkan *availability* bisa dilakukan dengan mengimplementasi *container*, melakukan *monitoring* pada *microservice* (Lyu et al., 2020), serta penggunaan *container orchestrator* (Vayghan et al., n.d.).

2.4. *Message Queue*

Message queue merupakan sistem yang mampu menampung data dalam bentuk antrian (*queue*). Sistem ini memungkinkan beberapa terminal untuk saling terhubung untuk saling mengirim dan menerima data. *Message queue* terdiri dari beberapa *worker machine* atau *broker* yang bekerja untuk mengirim dan menerima data. Entitas yang mengirim data disebut sebagai *producer*, dan entitas yang menerima data disebut sebagai *consumer*. Pesan atau data dikelompokkan dalam suatu topik atau antrian berdasarkan *producer* atau *consumer tertentu* (Fu et al., 2021).

Protokol komunikasi yang digunakan oleh *message queue* diantaranya *AMQP*, *XMPP*, *REST*, dan *STOMP*. *Message queue* mengolah pesan menggunakan mekanisme *push* dan *pull*. Mekanisme *push* merupakan mekanisme di mana *message queue* melakukan *push* terhadap data yang dikirimkan ke *consumer* secara terus menerus, sedangkan mekanisme *pull* merupakan mekanisme di mana *message queue* melakukan *pull* terhadap pesan yang diterima oleh *consumer* dalam rentang waktu tertentu. Ada dua model antrian yang digunakan dalam *message queue*, diantaranya *point-to-point (P2P)* dan *publish-subscribe (pubsub)*. Model *point-to-point* hanya mengizinkan sebuah *consumer* menerima data dari satu topik atau *queue* tertentu, sedangkan model *publish-subscribe* mengizinkan sebuah *consumer* menerima data lebih dari satu topik atau *queue* (Fu et al., 2021).

Beberapa pelaku industri menggunakan *message queue* untuk menjalankan aplikasinya

Tabel 1 *Message queue* yang digunakan di kalangan industri

<i>Message Queue Yang Digunakan</i>	<i>Pelaku Industri</i>	<i>Studi Kasus</i>
<i>NSQ</i>	Tokopedia	<i>Recommendation engine</i> ³
	<i>Bitly</i>	Menyelesaikan permasalahan <i>data loss</i> ⁴
<i>Kafka</i>	Gojek	<i>Merchant data</i> ⁵
	<i>Grab</i>	<i>Mission critical log, event sourcing dan stream processing</i> ⁶
	<i>Adidas</i>	<i>Real-time event processing, keperluan monitoring, analitik, serta laporan</i>

2.5. Container

Container merupakan implementasi dari teknologi virtualisasi untuk membangun, men-*deploy*, dan mengorkestrasi aplikasi (Yadav et al., 2019). Proses untuk membangun dan menggabungkan dependensi aplikasi, serta *library* sistem yang dibutuhkan oleh aplikasi disebut dengan *containerization*. Dibandingkan dengan *virtual machine*, teknologi *container* memiliki waktu *boot* yang lebih cepat, ukuran yang lebih kecil, penggunaan sumber daya pada *host* yang lebih hemat, serta *creation time* yang lebih cepat (Potdar et al., 2020). Setiap sistem operasi memiliki teknologi *container*, sebagai contoh sistem operasi *Linux* memiliki *container* seperti *Docker*, *LXC*, *Linux Cointainers*, dan *Open VZ* (Yadav et al., 2019).

³ <https://academy.tokopedia.com/tokolabs/user-based-recommendation-engine/#0>

⁴ <https://word.bitly.com/post/33232969144/nsq>

⁵ <https://medium.com/gojekengineering/a-smart-pipeline-for-merchant-data-2be48371cc87>

⁶ <https://kafka.apache.org/powered-by>

Docker merupakan salah satu teknologi *container* yang memastikan sebuah aplikasi dapat berjalan di berbagai *environment* serta mengotomasi aplikasi yang akan berjalan dalam *container* (Potdar et al., 2020). *Docker* memiliki empat komponen yang terdiri dari *Docker Container*, *Docker Client-Server*, *Docker Images*, dan *Docker engine*.

Yadav, dkk (2019) memaparkan perbedaan antara *virtual machine* dengan *hypervisor* sebagai berikut (Yadav et al., 2019).

Tabel 2 Perbedaan Virtual Machine dan Container

Faktor Pembeda	<i>Virtual Machine</i>	<i>Hypervisor</i>
Dukungan Sistem Operasi	Mebutuhkan sistem operasi guest	Dapat berbagi dengan sistem operasi host
Waktu Booting	Lambat	Cepat
Standarisasi	Sangat spesifik terhadap sistem operasi	Sangat spesifik terhadap aplikasi
Portabilitas	Kurang portabel	Lebih Portabel
Kebutuhan Server	Mebutuhkan lebih banyak server	Mebutuhkan lebih sedikit server
Keamanan	Bergantung pada hypervisor	Data dapat diakses di tingkat kernel, sehingga kurang aman
Redundansi	Informasi yang dimiliki lebih redundan, karena VM memiliki sistem operasi sendiri	Lebih sedikit informasi redundan, karena berbagi informasi pada sistem operasi yang sama
Akses Ke Perangkat Keras	Tidak memungkinkan	Memungkinkan
Distribusi Sumber Daya	Mebutuhkan lebih banyak sumber daya	Mebutuhkan lebih sedikit sumber daya

Kebutuhan Memori	Membutuhkan lebih banyak memori, karena setiap guest memiliki sistem operasi masing-masing	Membutuhkan lebih sedikit memori, karena berbagi dengan sistem operasi host
Berbagi File dan Library	Tidak memungkinkan	Memungkinkan

2.6. NoSQL

NoSQL (Not Only SQL) merupakan basis data yang tidak memiliki *schema* tetap sehingga mampu menyimpan data secara fleksibel dibandingkan dengan basis data *SQL* (Khan et al., 2022). *NoSQL* memiliki keuntungan diantaranya kecepatan, biaya, skalabilitas, fleksibilitas, dan kemudahan. Definisi, penggunaan, dan contoh basis data *NoSQL* sebagai berikut (Moniruzzaman & Hossain, 2013).

Tabel 3 Jenis-Jenis NoSQL

Jenis <i>NoSQL</i>	Mekanisme Penyimpanan	Penggunaan	Contoh
Key-Value	Berupa <i>key</i> dengan <i>value</i> yang berupa data sederhana seperti teks atau data kompleks seperti <i>set</i> atau <i>list</i>	Melakukan <i>retrieval</i> pada data seperti <i>session</i> atau <i>credential token</i>	Redis, DynamoDB, Voldemort, BerkeleyDB, Riak
Document	Data disimpan dalam pasangan <i>attribute</i> dan <i>value</i> , di mana dalam satu kolom bisa terdapat banyak <i>attribute</i>	Digunakan untuk menyimpan data literal seperti teks, <i>XML</i> , atau <i>JSON</i> , serta menyimpan data	CouchDB, MongoDB, Google BigTable

	dan data yang disimpan dalam satu baris bisa beragam	yang bersifat <i>NULLABLE</i>	
Wide Column	Data disimpan dalam kolom yang mampu menyimpan banyak atribut dalam sebuah <i>key</i> secara terdistribusi	Penyimpanan data terdistribusi yang membutuhkan mekanisme <i>versioning</i> dan sistem yang menggunakan <i>batch processing</i>	Google BigTable, Cassandra, Amazon SimpleDB, Amazon DynamoDB
Graph	Terdiri dari <i>node</i> yang merepresentasikan objek, <i>edge</i> yang merepresentasikan keterhubungan antar <i>node</i> , dan property yang dimiliki oleh masing-masing <i>node</i>	Mencari keterhubungan antara satu data dengan data lain	Neo4j, InfoGrid, Sones GraphDB, AllegroGraph, InfiniteGraph

2.7. *Performance Testing*

Performance testing dilakukan untuk menguji kinerja suatu aplikasi setelah aplikasi tersebut dirilis. Menurut (Wang & Wu, 2019) *performance testing* dibagi menjadi beberapa jenis, diantaranya:

1. *General Performance Testing*: menjalankan aplikasi dengan perangkat keras dan perangkat lunak tanpa dikenakan beban pada perangkat-perangkat tersebut
2. *Stability Testing*: menguji aplikasi secara berkelanjutan untuk menguji stabilitas aplikasi
3. *Load Testing*: menguji aplikasi secara berkelanjutan dengan Batasan beban tertentu untuk menguji stabilitas aplikasi. Pengujian ini ditujukan untuk mengetahui seberapa stabil suatu aplikasi pada kondisi kritis.
4. *Stress Testing*: menguji aplikasi secara berkelanjutan hingga aplikasi kolaps. Pengujian ini menambah beban pada aplikasi secara bertahap hingga aplikasi tidak mampu menerima beban yang dikirim. Tujuannya adalah untuk mengetahui performa maksimal sistem

Adapun indikator yang digunakan pada *performance testing* adalah sebagai berikut (Fu et al., 2021)(Wang & Wu, 2019):

1. *Response Time*: Waktu yang dibutuhkan oleh sistem untuk menerima *response*
2. *Concurrent Users*: Jumlah pengguna yang mengakses aplikasi secara bersamaan dalam satu waktu
3. *Throughput*: Jumlah *request* dari *user* yang diproses oleh aplikasi dalam satuan waktu tertentu. Semakin tinggi *throughput* artinya data pada aplikasi tersebut cenderung untuk tidak di-*backlog*.

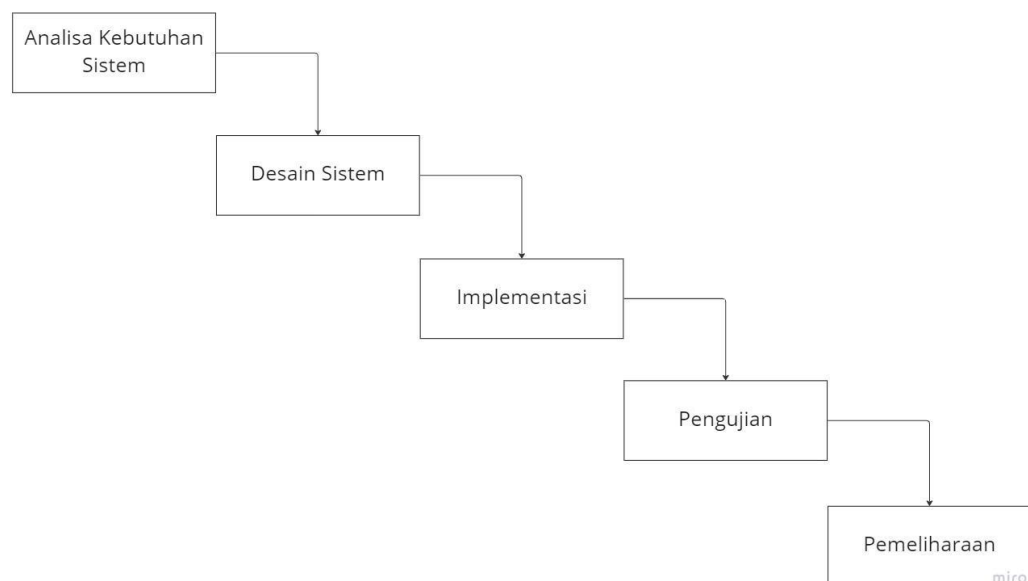
4. *Latency*: Waktu yang diperlukan *request* untuk dapat diterima oleh aplikasi. Semakin rendah *latency*, artinya aplikasi mampu memproses data volume besar dengan *delay* yang minim.
5. *Utilization*: Mengukur penggunaan sumber daya sistem, seperti berapa banyak *RAM* yang digunakan dan berapa persen *CPU* yang digunakan.

BAB 3

METODOLOGI PENELITIAN

Pada bab ini dijelaskan tentang metodologi penelitian yang digunakan. Penelitian ini bertujuan untuk membangun perangkat lunak, oleh karena itu metodologi yang digunakan adalah daur hidup pengembangan perangkat lunak (*software development life cycle*) dengan metode *waterfall*. Metodologi ini dipilih karena proses pengembangan perangkat lunak dikerjakan oleh tim dalam skala kecil dan tidak membutuhkan umpan balik dari pengguna. Dalam penelitian ini, metodologi tersebut terbagi menjadi lima tahapan diantaranya: analisis kebutuhan sistem, desain sistem, implementasi, pengujian, dan pemeliharaan.

Gambar 1 Diagram Alir Penelitian



3.1. Analisis Kebutuhan Sistem

Berdasarkan komponen dari *microservice* dan cara untuk mencapai *high availability*, sistem pada penelitian ini dirancang menggunakan *message queue* dengan arsitektur *microservice*. *Message queue* digunakan untuk memisahkan ketergantungan antaraplikasi, dengan kata lain jika salah satu aplikasi mengalami *downtime*, maka aplikasi lain tetap dapat berjalan.

Selain itu, *message queue* dipilih karena Sistem Deteksi Pemalsuan Dokumen membutuhkan waktu paling lama 2 menit untuk memeriksa sebuah dokumen. Dengan memanfaatkan pola komunikasi asinkron yang disediakan oleh *message queue*, maka *client* tidak perlu menunggu *request* selesai diproses. Untuk mendukung *high availability* pada sistem, spesifikasi *message queue* yang digunakan serta cara masing-masing *message queue* untuk mencapai *high availability* sebagai berikut.

Tabel 4 Spesifikasi Message Queue Yang Digunakan

<i>Message Queue</i>	Protokol Komunikasi	Cara Mencapai High Availability
<i>Kafka</i>	<i>TCP</i>	Men-deploy banyak <i>nsqlookupd instance</i> ⁷
<i>RabbitMQ</i>	<i>AMQP, MQTT, STOMP, HTTP, Websocket</i>	<i>Queue quorum</i> ⁸
<i>NSQ</i>	<i>TCP</i>	<i>Partition</i> ⁹

Di samping itu, menurut, (Alexandrov & Dimov, 2013) melakukan pemantauan (*monitoring*) pada sistem menjadi salah satu untuk mencapai *high availability*.

Proses pengembangan aplikasi ini menggunakan *container*. Hal ini dikarenakan waktu yang dibutuhkan *container* ketika menjalankan dependensi lebih cepat dibandingkan *virtual machine*, sehingga dapat mengurangi *downtime* ketika aplikasi perlu diperbaiki atau ada pembaruan pada aplikasi tersebut. Selain itu, *container* digunakan untuk memasang dependensi-dependensi yang dibutuhkan.

⁷ <https://nsq.io/overview/design.html#eliminating-spofs>

⁸ <https://www.rabbitmq.com/quorum-queues.html#availability>

⁹ <https://www.redhat.com/en/resources/high-availability-for-apache-kafka-detail>

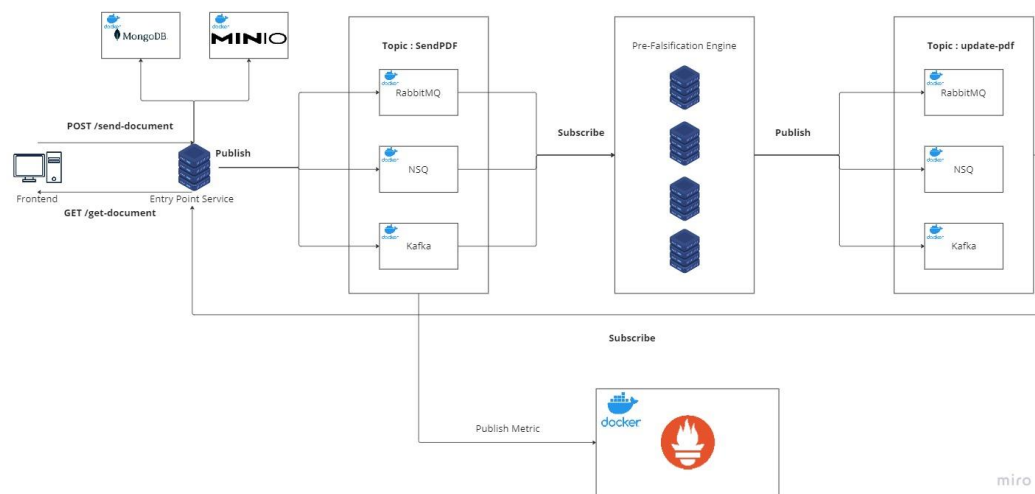
3.2. Desain Sistem

Tahap desain sistem merupakan tahapan di mana rancangan terhadap arsitektur sistem, skema basis data, dan logika bisnis dari aplikasi.

3.2.1 Arsitektur Sistem

Penelitian ini menggunakan *container* untuk membangun dan *deploy* aplikasi. Berikut arsitektur sistem yang diajukan.

Gambar 2 Desain Sistem



Terdapat *Entry Point Service* yang berfungsi untuk menerima *input* berupa dokumen dengan format PDF. Informasi dari dokumen tersebut akan disimpan dalam *object storage* dan *document storage* yaitu *MinIO* dan *MongoDB*. Data yang disimpan dalam *MinIO* adalah dokumen PDF asli, sedangkan data yang disimpan dalam *MongoDB* adalah hasil evaluasi dari Sistem Deteksi Pemalsuan Dokumen yang kemudian disebut dengan *Pre-Falsification Engine*. Kemudian terdapat *Pre-Falsification Engine* yang berfungsi untuk memeriksa apakah pada dokumen yang dikirim terdapat pemalsuan dokumen. Selanjutnya, terdapat *message queue* yang bertugas melakukan *publish* dan *subscribe* terhadap topik tertentu. Kriteria *message queue* yang digunakan pada penelitian ini adalah *message queue* yang mendukung *high availability* dengan rincian sebagai berikut.

3.2.2 Skema Basis Data

Skema basis data digunakan sebagai acuan format penyimpanan data dari aplikasi ke basis data. Pada penelitian ini, skema basis data yang dibuat adalah skema dokumen untuk *MongoDB* dengan format *JSON* (*JavaScript Object Notation*). Skema dokumen tersebut sebagai berikut.

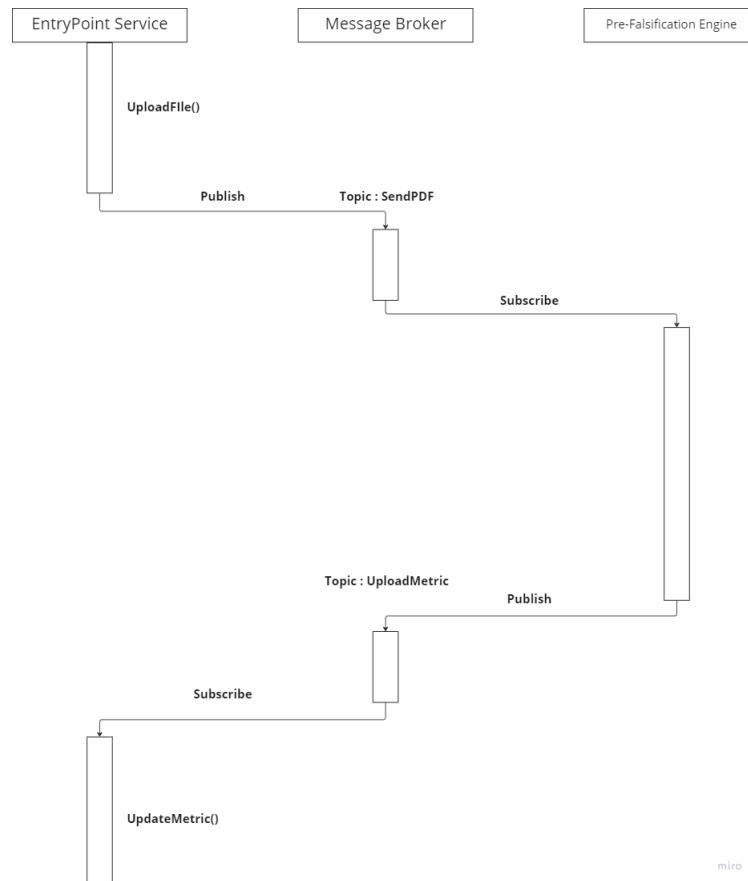
Tabel 5 Skema Dokumen

Properti	Tipe Data	Keterangan
FileObjectID	<i>String</i>	Nilai property object_id yang dibuat secara otomatis oleh <i>MongoDB</i>
Filename	<i>String</i>	Nama file yang dibuat secara acak menggunakan <i>UUID</i>
Timestamp	<i>String</i>	Penanda waktu kapan dokumen diunggah

3.2.3 Logika Bisnis

Logika bisnis dari aplikasi dirancang untuk menerjemahkan logika aplikasi ke dalam bentuk bisnis sehingga lebih mudah dipahami bagi orang awam. Pada penelitian ini, logika bisnis disajikan dalam *sequence diagram* sebagai berikut.

Gambar 3 Sequence Diagram Sistem



3.3. Implementasi

Hasil akhir dari analisis kebutuhan sistem dan desain sistem adalah kode program yang dapat berjalan dengan baik sesuai kebutuhan sistem dan kebutuhan bisnis. Perangkat lunak yang digunakan untuk mengimplementasi kebutuhan bisnis dan kebutuhan sistem adalah sebagai berikut

Tabel 6 Perangkat Lunak Yang Digunakan Untuk Pengembangan

Sistem Operasi	<i>Windows 10</i>
<i>Golang</i>	Versi 1.19
<i>Docker</i>	Versi 4.12

3.4. Pengujian

Tahapan pengujian dilakukan untuk menguji fungsionalitas serta kinerja sistem berdasarkan kebutuhan bisnis dan kebutuhan sistem. Metrik yang digunakan untuk mengevaluasi kinerja sistem serta skenario pengujian adalah sebagai berikut.

3.4.1 Metrik Evaluasi

Pada metrik pengujian (*evaluation metric*) yang digunakan adalah *throughput* dan *latency*. *Throughput* dan *latency* didefinisikan pada persamaan berikut

$$Throughput = \frac{\text{Jumlah message yang dikirim}}{\text{waktu yang diperlukan untuk mengirim message (detik)}}$$

$$Latency = \text{waktu ketika request selesai} - \text{waktu ketika request dimulai}$$

Selain *throughput* dan *latency*, metrik lain digunakan adalah penggunaan memori dari masing-masing *container message broker* selama *load testing* berjalan.

3.4.2 Skenario Pengujian

Proses pengujian sistem menggunakan skenario *load testing* dengan membandingkan performa sistem *entry point service* terhadap *message queue* dan *REST API*. Pengujian sistem menggunakan bantuan aplikasi *Apache JMeter*. Parameter yang digunakan untuk keperluan *load testing* pada aplikasi tersebut adalah sebagai berikut.

Tabel 7 Parameter Pengujian

Parameter	Tipe Data	Keterangan
<i>Number of Threads (users)</i>	<i>Integer</i>	Jumlah user pada simulasi
<i>Loop Count</i>	<i>Integer</i>	Jumlah iterasi yang dilakukan oleh sebuah thread
<i>Ramp Period</i>	<i>Integer</i>	Waktu yang diperlukan untuk membuat sebuah <i>thread</i> .

Untuk mensimulasikan aktivitas *user*, sistem *backend pre-plagiarism checker* diuji dengan skenario *single thread multiple loop*, *single loop multiple thread* dan *multiple thread multiple loop*, di mana *thread* mensimulasikan *user* dan *loop* mensimulasikan berapa kali seorang *user* mengakses aplikasi tersebut. *Ramp period* pada penelitian ini dikunci di 180 detik.

Perangkat keras yang digunakan untuk pengujian adalah sebagai berikut

Tabel 8 Perangkat Keras Yang Digunakan Untuk Pengujian

Perangkat Keras	Kapasitas
<i>Processor</i>	Intel® Core™ i9
<i>Memory</i>	32 GB DDR4
<i>Storage</i>	512 GB SSD NVMe

3.5. Pemeliharaan

Aplikasi yang sudah berjalan dengan baik di *local environment* dan dinyatakan layak untuk dirilis, kemudian memasuki *production environment*. Di *production environment*, aplikasi harus diobservasi untuk mengetahui kinerja aplikasi ketika digunakan langsung oleh pengguna. Pemeliharaan aplikasi meliputi *observability*, *tracing*, *monitoring*, serta penambahan dan perbaikan pada rilis di masa depan.

BAB 4

HASIL DAN PEMBAHASAN

4.1. Hasil Implementasi Sistem

Implementasi dari kebutuhan bisnis dan kebutuhan sistem adalah sebagai berikut

4.1.1. *Entry Point Service*

Secara keseluruhan, sistem pada penelitian ini berada pada *entry point service*. Aplikasi tersebut dibangun menggunakan bahasa Go (Golang) dengan dependensi berupa *package* pendukung dengan rincian sebagai berikut:

- *HTTP Server: gorilla mux*
- *Message Queue Driver: go-nsq, amqp, sarama*
- *Secret Management: godotenv*
- *Database Driver: minio-go, mongo-driver*
- *UUID Generator: uuid*

Pada *service* ini, terdapat *REST endpoint* yang mengintegrasikan proses *input* dari *user/frontend* dengan *message queue*. Adapun ketentuan *request* dan *response* dari masing-masing *endpoint* sebagai berikut.

Tabel 9 API Contract

<i>Endpoint</i>	<i>HTTP Method</i>	<i>Request Body</i>	<i>Request Type</i>	<i>URL Parameter</i>
send-document	<i>POST</i>	<i>file</i>	<i>File</i>	-
show-document	<i>GET</i>	-	-	<i>document-name</i>

4.1.2. File Docker Compose

Sistem pada penelitian ini menggunakan *Docker* untuk memenuhi dependensi aplikasi. Dependensi-dependensi tersebut berupa *image* yang diunduh dari *Docker Hub*. Berikut *image*, versi, serta jenis *image* yang digunakan untuk membangun sistem *backend Pre-Falsification* pada penelitian ini.

Tabel 10 Daftar *Docker Image* Yang Digunakan

<i>Image</i>	<i>Versi</i>	<i>Jenis</i>
<i>MinIO</i>	<i>latest</i>	<i>Object Storage</i>
<i>MongoDB</i>	6.0.2	<i>Database</i>
<i>NSQ</i>	<i>latest</i>	<i>Message Queue</i>
<i>RabbitMQ</i>	3-management	<i>Message Queue</i>
<i>Kafka</i>	3.2-debian-11	<i>Message Queue</i>
<i>ZooKeeper</i>	3.8-debian-11	<i>Daemon</i>
<i>Prometheus</i>	v2.39.0	<i>Instrumentation</i>

Seluruh *image* dikompilasi dalam *file docker compose*, di mana *container* dari masing-masing *image* bisa langsung dibuat lewat *file* tersebut. *Docker compose* juga bisa digunakan untuk mengatur *environment variable* pada *container*, membuat dan mengatur *network* serta *port* supaya setiap *container* dapat saling berkomunikasi, dan membuat *command* yang akan langsung dijalankan ketika *container* dibuat.

4.1.3. Instrumentasi

Instrumentasi adalah proses untuk melakukan observasi terhadap sistem saat sistem sedang berjalan. Instrumentasi terhadap sistem dapat dilakukan secara statis (menganalisa kode dan menyisipkan kode instrumentasi) atau secara dinamis (memanipulasi sistem ketika sistem berjalan dengan cara menyisipkan kode instrumentasi) (Tchamgoue & Fischmeister, 2016).

Instrumentasi pada sistem backend pre-plagiarism checker dilakukan secara statis menggunakan Prometheus dengan cara menyisipkan kode instrumentasi pada kode yang berhubungan dengan *message queue*.

Tujuannya untuk mengetahui metrik yang dihasilkan ketika aplikasi sedang berjalan. Jenis metrik yang digunakan pada sistem adalah *counter* dan *histogram*. *Counter* adalah jenis metrik yang nilainya terus bertambah, sedangkan *histogram* adalah kumpulan hasil observasi yang disimpan dalam keranjang (*bucket*) yang bisa dikonfigurasi. Implementasi dari metrik tersebut sebagai berikut, di mana *x* adalah nama dari *message queue*. *Counter* digunakan untuk mengukur *throughput*, sedangkan *histogram* digunakan untuk mengukur *latency*.

Tabel 11 Metrik *Prometheus* Yang Digunakan

Metrik	Jenis Metrik	Keterangan
<i>x_message_pumped_count</i>	<i>Counter</i>	Jumlah <i>message</i> yang dikirim ke <i>message queue</i> oleh <i>service</i>
<i>x_latency_seconds</i>	<i>Histogram</i>	Waktu yang diperlukan <i>message queue</i> untuk menerima <i>message</i>
<i>Container_memory_usage_bytes{name="x"}</i>	<i>Gauge</i>	Jumlah memori yang digunakan oleh <i>container</i>

4.2. Hasil Pengujian Sistem

Pada penelitian ini, proses instrumentasi dilakukan pada *entry point service*. Metrik berupa rata-rata jumlah *message* yang dikirim dari *entry point service* ke *message queue* dalam satu menit (*throughput*) dan waktu yang diperlukan *message queue* untuk menerima *message* (*latency*) diukur pada *service* ini. *Latency* diukur selama proses pengiriman *message* dari *entry point service* ke *message queue*. Selain itu, metrik berupa penggunaan memori pada *container* juga diperhitungkan untuk mengetahui berapa banyak memori yang digunakan selama pengujian. Metrik penggunaan *container* dihitung ketika *throughput* dari *message queue* mencapai nilai tertinggi. Berdasarkan skenario pengujian sistem, didapat hasil *evaluation metric* dari masing-masing *message queue* sebagai berikut.

4.2.1. *Single Thread, Multiple Loop*

Tabel 12 Hasil Pengujian Skenario *Single Thread, Multiple Loop*

<i>Message queue / REST</i>	<i>Jumlah Loop</i>	<i>Throughput (message / min)</i>	<i>Min Latency (second)</i>	<i>Max Latency (second)</i>	<i>Memory Usage (MB)</i>
<i>NSQ</i>	100	1,779	0,004	0,005	13,706
	1000	17,603	0,003	0,004	74,473
	10000	29,649	0,003	0,005	155,418
<i>RabbitMQ</i>	100	1,749	0,031	0,032	146,026
	1000	14,350	0,027	0,028	152,936
	10000	14,280	0,027	0,029	153,341
<i>Kafka</i>	100	1,802	0,003	0,004	424,505
	1000	17,402	0,001	0,002	475,770
	10000	32,105	0,001	0,002	442,277
<i>REST Client</i>	100	1,731	0,002	0,002	59,719
	1000	17,261	0,002	0,003	420,368
	10000	56,491	0,001	0,002	1474,461

4.2.2. Single Loop, Multiple Thread

Tabel 13 Pengujian Skenario *Single Loop, Multiple Thread*

<i>Message Queue/ REST</i>	<i>Jumlah Thread</i>	<i>Throughput (message / min)</i>	<i>Min Latency (second)</i>	<i>Max Latency (second)</i>	<i>Memory Usage (MB)</i>
<i>NSQ</i>	100	0,561	0,002	0,005	9,302
	1000	5,561	0,003	0,004	35,520
	10000	55,614	0,004	0,009	231,813
<i>RabbitMQ</i>	100	0,561	0,020	0,053	156,618
	1000	5,561	0,028	0,029	154,759
	10000	55,596	0,026	0,032	159,023
<i>Kafka</i>	100	0,561	0,002	0,018	393,297
	1000	5,561	0,001	0,008	408,088
	10000	55,807	0,001	0,006	465,010
<i>REST Client</i>	100	0,561	0,001	0,004	28,966
	1000	5,561	0,001	0,001	144,510
	10000	56,508	0,004	0,008	2934,771

4.2.3. Multiple Thread, Multiple Loop

Tabel 14 Hasil Pengujian Skenario *Multiple Loop Multiple Thread*

<i>Message queue/ REST</i>	Jumlah Thread	Jumlah Loop	<i>Throughput (message / min)</i>	<i>Min Latency (second)</i>	<i>Max Latency (second)</i>	<i>Memory Usage (MB)</i>
<i>NSQ</i>	50	100	27,842	0,003	0,004	213,954
	100	50	27,877	0,007	0,008	138,768
<i>RabbitMQ</i>	50	100	27,929	0,024	0,027	223,715
	100	50	27,859	0,025	0,027	171,589
<i>Kafka</i>	50	100	27,947	0,001	0,003	420,712
	100	50	27,912	0,001	0,002	435,580
<i>REST Client</i>	50	100	28,069	0,002	0,002	770,449
	100	50	27,877	0,002	0,003	724,000

4.3. Analisis Implementasi & Hasil Pengujian Sistem

Hasil implementasi berupa sistem yang dikembangkan pada penelitian ini telah mampu memisahkan dependensi antara *input user* dengan *pre-plagiarism engine* menggunakan *message queue*. Di samping itu, sistem yang dikembangkan juga mampu mengirim data ke *pre-plagiarism engine* menggunakan *REST client*.

Berdasarkan hasil pengujian sistem, *throughput* dari sistem *backend* ke seluruh *message queue* dan *REST client* hampir seragam. Penulis menduga bahwa arsitektur maupun protokol komunikasi yang digunakan oleh *message queue* tidak berpengaruh terhadap *throughput* sistem. Namun, *throughput* pada *REST client* bisa menurun jika waktu yang dibutuhkan oleh *REST server* untuk memproses *request* lebih lama. Di samping itu, hasil

pengujian pada skenario *single loop multiple thread* menunjukkan *throughput* yang hampir seragam untuk setiap *test case* dan setiap *message queue*. Hal ini dikarenakan sistem mampu menangani *request* secara *concurrent*, berbeda dengan hasil skenario *single thread multiple loop* di mana hasil tersebut menunjukkan bahwa sistem yang dikembangkan belum mampu menangani *multiple payload* dari sebuah *thread* secara *concurrent*.

Dari segi *latency*, di antara seluruh *message queue* yang digunakan, *RabbitMQ* memiliki *latency* yang tinggi. Artinya data dari *entry point service* membutuhkan waktu yang lebih lama untuk diterima oleh *RabbitMQ*. Hal ini disebabkan oleh protokol pengiriman data yang digunakan *RabbitMQ*. Pada penelitian ini, protokol yang digunakan oleh *RabbitMQ* adalah *AMQP* versi 0-9-1. Berdasarkan spesifikasi *AMQP* versi 0-9-1¹⁰, proses yang terjadi dari pengiriman hingga penerimaan pesan adalah sebagai berikut:

1. *Client* membuka koneksi *TCP/IP* ke *server* dan mengirim *protocol header*.
2. *Server* merespon dengan mengirim versi protokol serta mekanisme keamanan yang digunakan *server*.
3. *Client* memilih mekanisme keamanan yang dikirim *server*.
4. *Server* memulai proses otentikasi
5. *Client* mengirim respon otentikasi, seperti mengirimkan *username* dan *password*.
6. *Client* membuka koneksi dan memilih *virtual host*.
7. *Server* mengonfirmasi dan memvalidasi *virtual host*.
8. *Client* dapat mengirim data hingga *client* atau *server* memutuskan koneksi menggunakan mekanisme *handshake*.

¹⁰ <https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf>

Hal ini berbanding terbalik dengan *TCP* yang menggunakan *three-way handshake*¹¹ dengan alur sebagai berikut:

1. *Client* mengirim *flag SYN* dengan *sequence number x*
2. *Server* menerima *flag SYN*, dan mengirimkan *flag SYN,ACK* ke *client* beserta *payload* berupa *sequence number y* dan *sequence number x + 1*.
3. *Client* menerima *flag SYN* dari *server* dengan *sequence number y*. Kemudian *client* mengirim *flag ACK* ke *server*.
4. *Server* menerima *flag ACK*, dan koneksi sudah terbentuk.

Di samping itu, *AMQP* menggunakan *TCP* untuk menyambungkan *client/producer* ke *AMQP server*, tidak seperti *NSQ* atau *Kafka* yang langsung menggunakan *TCP* supaya *client/producer* bisa berkomunikasi dengan *server*.

Pada sisi konsumsi memori, *REST client* memiliki konsumsi memori tertinggi dibandingkan *message queue*. Sementara itu dari seluruh *message queue*, *RabbitMQ* memiliki konsumsi memori yang stabil, dan *Kafka* memiliki konsumsi memori tertinggi. Konsumsi memori oleh *Kafka* yang tinggi dikarenakan *Kafka* dibangun di atas *JVM* (*Java Virtual Machine*), di mana salah satu fitur bawaan dari *JVM* adalah *garbage collector*. Sementara itu, konsumsi memori yang tinggi pada *REST client* disebabkan oleh proses yang dilakukan oleh *REST client* secara sinkron, mulai dari menerima data hingga menunggu respon balikan dari *REST server* sehingga membebani *REST client*.

Pada penelitian ini, kondisi *REST server* disimulasikan mampu menyelesaikan pekerjaan dalam rentang waktu 0 hingga 1 detik, sehingga menghasilkan metrik di atas. Namun, hasil metrik dapat berbeda jika kenyataannya *REST server* memerlukan waktu lebih lama dari yang disimulasikan pada peneltiain ini.

¹¹ <https://www.rfc-editor.org/rfc/rfc9293#name-initial-sequence-number-sel>

BAB 5

KESIMPULAN

5.1. Kesimpulan

Berdasarkan hasil implementasi dan hasil pengujian sistem, dapat diperoleh kesimpulan sebagai berikut:

1. Sistem yang dikembangkan pada penelitian ini mampu memisahkan proses *input user* dengan *pre-plagiarism engine* menggunakan *message queue* maupun *REST client/server*.
2. Berdasarkan hasil pengujian sistem, *RabbitMQ* memiliki konsumsi memori yang stabil untuk setiap skenario pengujian, dan *Kafka* memiliki *latency* paling rendah. Pada sisi *latency*, *message queue* dengan protokol komunikasi *TCP* memiliki performa lebih baik dibandingkan dengan *message queue* dengan protokol komunikasi *AMQP*, dan *throughput* pada sistem lebih baik menggunakan pola komunikasi asinkron (*message queue*) dibandingkan dengan sinkron (*REST*)

5.2. Saran

Berdasarkan hasil penelitian ini, saran yang dapat diberikan untuk penelitian di masa depan diantaranya

1. Melakukan pengembangan menggunakan *container orchestrator* seperti *Kubernetes*, *Nomad*, atau *Docker Swarm*
2. Melakukan teknik lain untuk mendukung *high availability* dengan cara mengimplementasikan *load balancing* atau *vertical scaling*
3. Menggunakan cara lain untuk komunikasi antar-service seperti *RPC* atau *GraphQL*

DAFTAR PUSTAKA

- Alexandrov, T., & Dimov, A. (2013). Software availability in the cloud. *ACM International Conference Proceeding Series*, 767, 193–200. <https://doi.org/10.1145/2516775.2516814>
- Berenberg, A., & Calder, B. (2023). Deployment Archetypes for Cloud Applications. *ACM Computing Surveys*, 55(3), 1–48. <https://doi.org/10.1145/3498336>
- Blas, G. (2019). *A Comparison of Message Brokers for Telerehabilitation Systems*. 1–10.
- Blinowski, G., Ojdowska, A., & Przybylek, A. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*, 10, 20357–20374. <https://doi.org/10.1109/ACCESS.2022.3152803>
- Fishman, T. (2009). We know it when we see it is not good enough: toward a standard definition of plagiarism that transcends theft, fraud, and copyright. In Proceedings of the Fourth Asia Pacific Conference on Educational Integrity (4APCEI). *University of Wollongong, NSW, Australia*. 2009, September, 1–5. http://www.opsi.gov.uk/RevisedStatutes/Acts/ukpga/1968/cukpga_19680060_en_1#pb1-11g1%0Ahttp://ro.uow.edu.au/cgi/viewcontent.cgi?article=%0A1037&context=apcei
- Fu, G., Zhang, Y., & Yu, G. (2021). A Fair Comparison of Message Queuing Systems. *IEEE Access*, 9(2), 421–432. <https://doi.org/10.1109/ACCESS.2020.3046503>
- Gholamian, S., & Ward, P. A. S. (2021). What Distributed Systems Say: A Study of Seven Spark Application Logs. *Proceedings of the IEEE Symposium on Reliable Distributed Systems, 2021-Septe*, 222–232. <https://doi.org/10.1109/SRDS53918.2021.00030>
- Hiraman, B. R., Viresh, M. C., & Abhijeet, C. K. (2018). A Study of Apache Kafka in Big Data Stream Processing. *2018 International Conference*

- on Information, Communication, Engineering and Technology, *ICICET 2018*, 1–3. <https://doi.org/10.1109/ICICET.2018.8533771>
- ind, Karambir, S. T. (2015). A Simulation Model for the Spiral Software Development Life Cycle. *International Journal of Innovative Research in Computer and Communication Engineering*, 03(05), 3823–3830. <https://doi.org/10.15680/ijircce.2015.0305013>
- John, V., & Liu, X. (2017). *A Survey of Distributed Message Broker Queues*. <http://arxiv.org/abs/1704.00411>
- Karabey Aksakalli, I., Çelik, T., Can, A. B., & Tekinerdoğan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software*, 180. <https://doi.org/10.1016/j.jss.2021.111014>
- Khan, W., Kumar, T., Cheng, Z., Raj, K., Roy, A. M., & Luo, B. (2022). *SQL and NoSQL Databases Software architectures performance analysis and assessments -- A Systematic Literature review*.
- Krzemien, W., Stagni, F., Haen, C., Mathe, Z., McNab, A., & Zdybal, M. (2019). Addressing Scalability with Message Queues: Architecture and Use Cases for DIRAC Interware. *EPJ Web of Conferences*, 214, 03018. <https://doi.org/10.1051/epjconf/201921403018>
- Lyu, Z., Wei, H., Bai, X., & Lian, C. (2020). Microservice-Based Architecture for an Energy Management System. *IEEE Systems Journal*, 14(4), 5061–5072. <https://doi.org/10.1109/JSYST.2020.2981095>
- Maccormack, A., Rusnak, J., & Baldwin, C. (2007). *Exploring the Duality between Product and Organizational Architectures: A Test of the “Mirroring” Hypothesis*.
- Mahmud, S., Bretag, T., & Foltýnek, T. (2019). Students’ Perceptions of Plagiarism Policy in Higher Education: a Comparison of the United Kingdom, Czechia, Poland and Romania. *Journal of Academic Ethics*, 17(3), 271–289. <https://doi.org/10.1007/s10805-018-9319-0>
- Márquez, G., & Astudillo, H. (2019). Identifying availability tactics to support security architectural design of microservice-based systems.

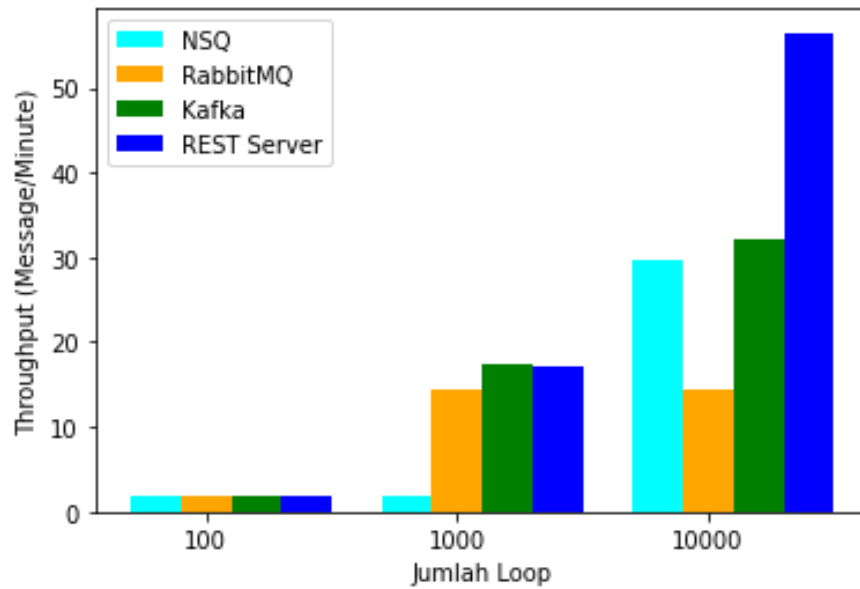
- ACM International Conference Proceeding Series*, 2, 123–131.
<https://doi.org/10.1145/3344948.3344996>
- Moniruzzaman, A. B. M., & Hossain, S. A. (2013). *NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison*. June.
- Moon, J. H., & Shine, Y. T. (2020). A study of distributed SDN controller based on apache kafka. *Proceedings - 2020 IEEE International Conference on Big Data and Smart Computing, BigComp 2020*, 44–47. <https://doi.org/10.1109/BigComp48618.2020.0-101>
- Potdar, A. M., Narayan, D. G., Kengond, S., & Mulla, M. M. (2020). Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, 171(2019), 1419–1428. <https://doi.org/10.1016/j.procs.2020.04.152>
- Sasidharan, R. (2022). *Implementation of High Available and Scalable Syslog Server with NoSQL Cassandra Database and Message Queue*. 9(1), 1–7. <https://doi.org/10.5923/j.ajca.20220901.01>
- Shafabakhsh, B., Lagerström, R., & Hacks, S. (2020). *Evaluating the Impact of Inter Process Communication in Microservice Architectures*. <http://ceur-ws.org>
- Tchamgoue, G. M., & Fischmeister, S. (2016). Lessons learned on assumptions and scalability with time-aware instrumentation. *Proceedings of the 13th International Conference on Embedded Software, EMSOFT 2016*. <https://doi.org/10.1145/2968478.2975584>
- Vayghan, L. A., Saied, M. A., Toeroe, M., & Khendek, F. (n.d.). *Kubernetes as an Availability Manager for Microservice Applications*.
- Wang, J., & Wu, J. (2019). Research on performance automation testing technology based on JMeter. *Proceedings - 2019 International Conference on Robots and Intelligent System, ICRIS 2019*, 55–58. <https://doi.org/10.1109/ICRIS.2019.00023>
- Wang, J., Yang, Y., Wang, T., Simon Sherratt, R., & Zhang, J. (2020). Big data service architecture: A survey. *Journal of Internet Technology*, 21(2), 393–405. <https://doi.org/10.3966/160792642020032102008>

- Waseem, M., Liang, P., Shahin, M., di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182. <https://doi.org/10.1016/j.jss.2021.111061>
- Yadav, A. K., Garg, M. L., & Ritika. (2019). Docker containers versus virtual machine-based virtualization. In *Advances in Intelligent Systems and Computing* (Vol. 814). Springer Singapore. https://doi.org/10.1007/978-981-13-1501-5_12

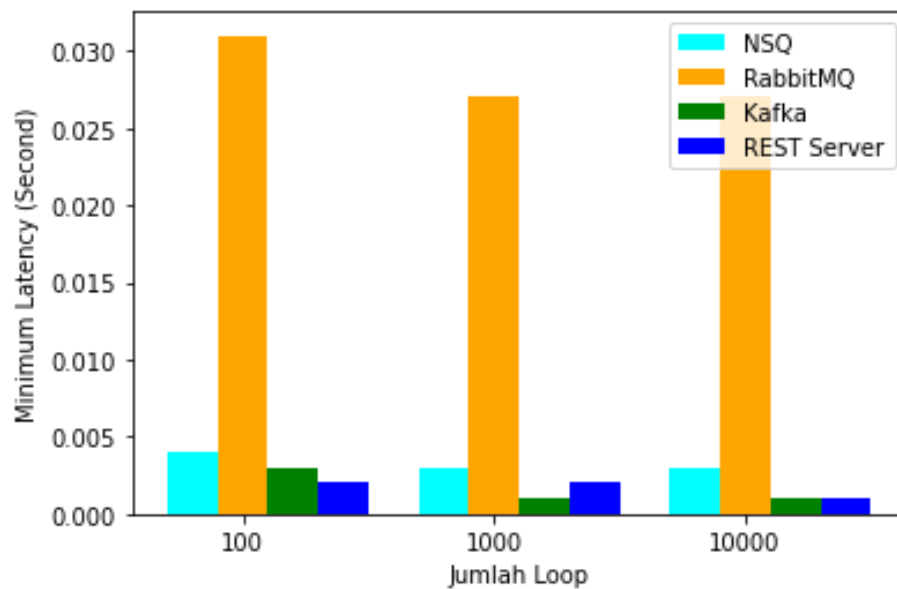
LAMPIRAN

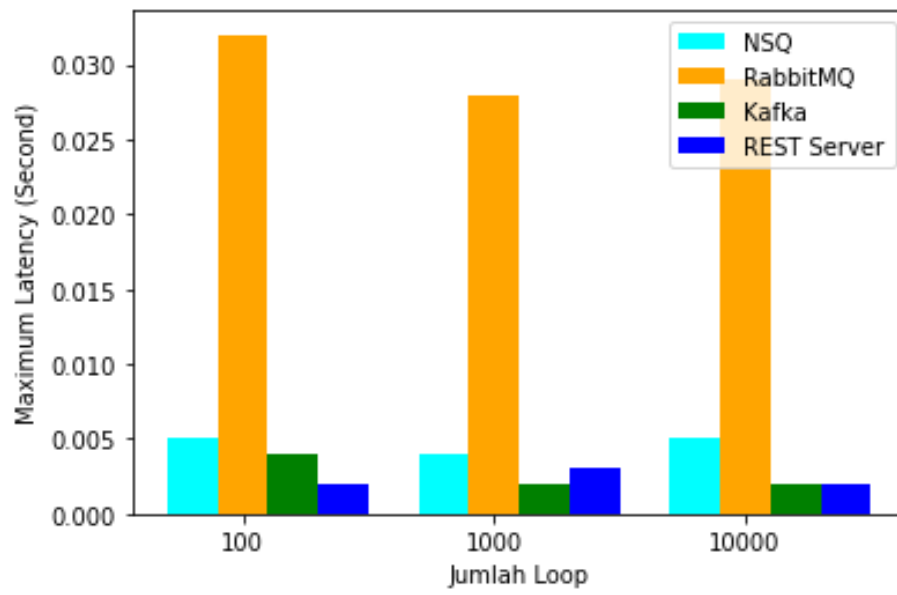
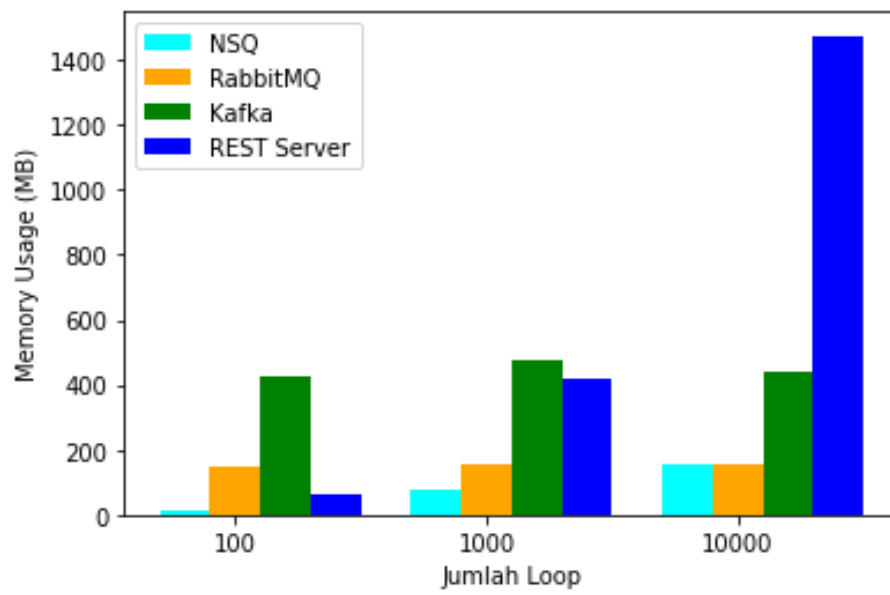
Lampiran 1. Hasil Pengujian Skenario Single Thread, Multiple Loop

Throughput



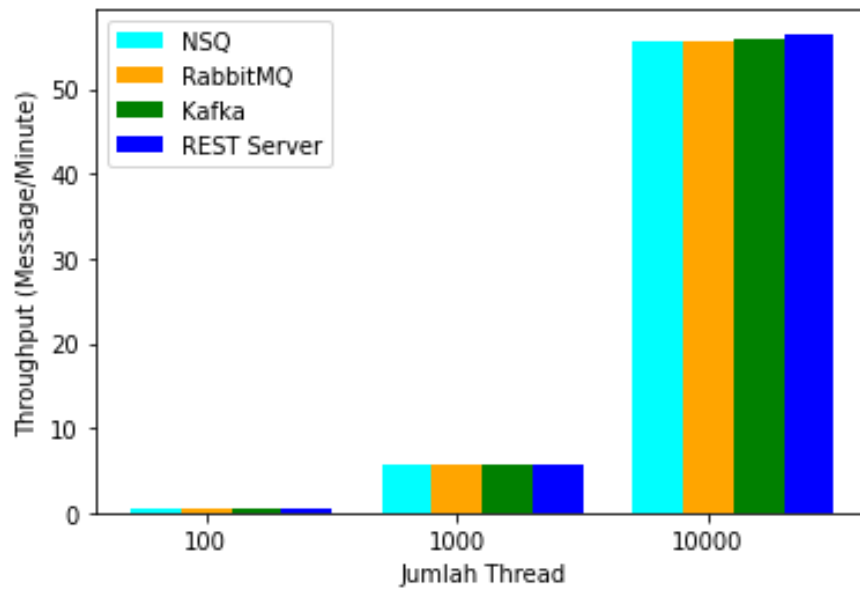
Minimum Latency



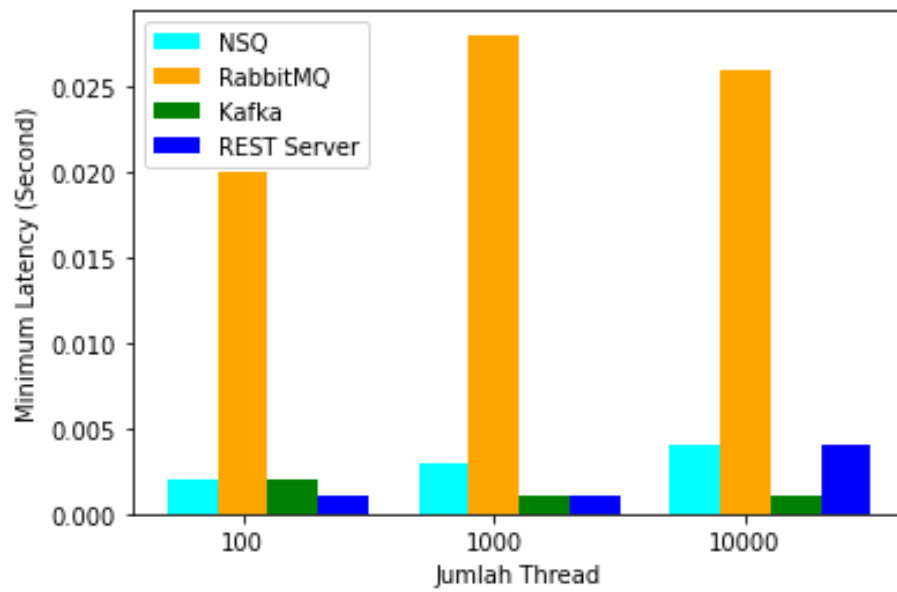
Maximum Latency*Memory Usage*

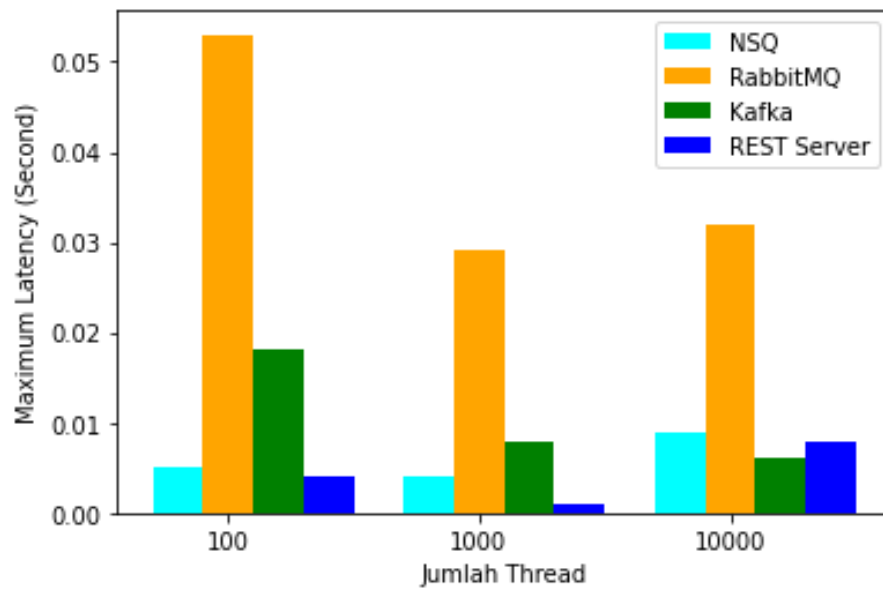
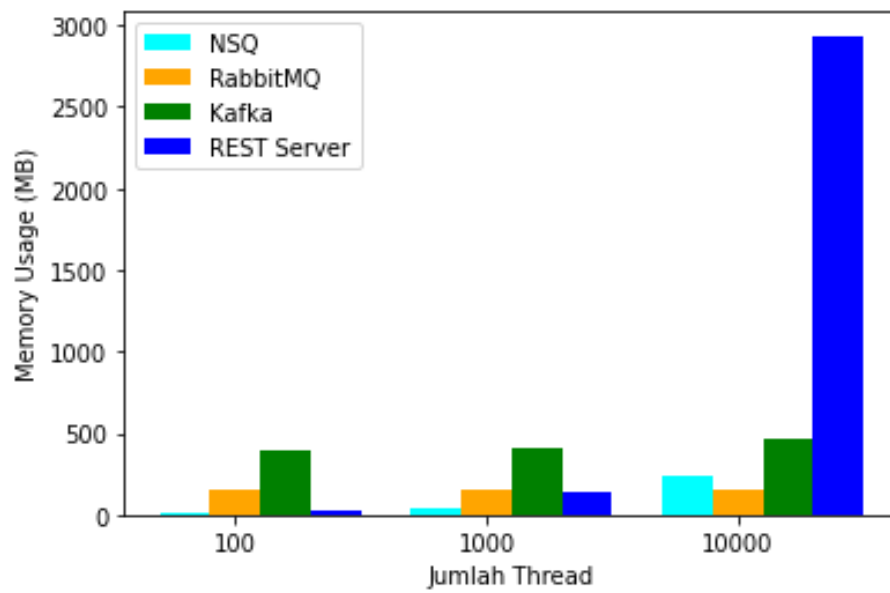
Lampiran 2. Hasil Pengujian Skenario Single Loop, Multiple Thread

Throughput



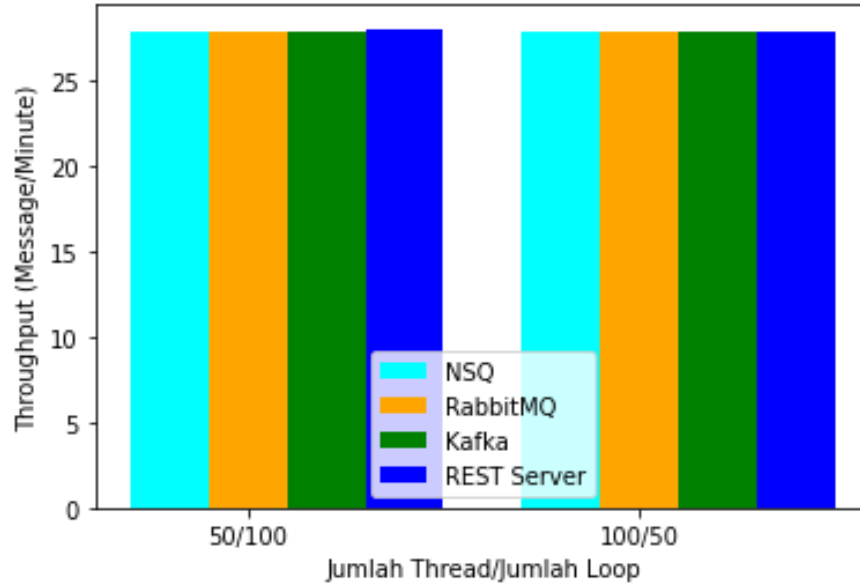
Minimum Latency



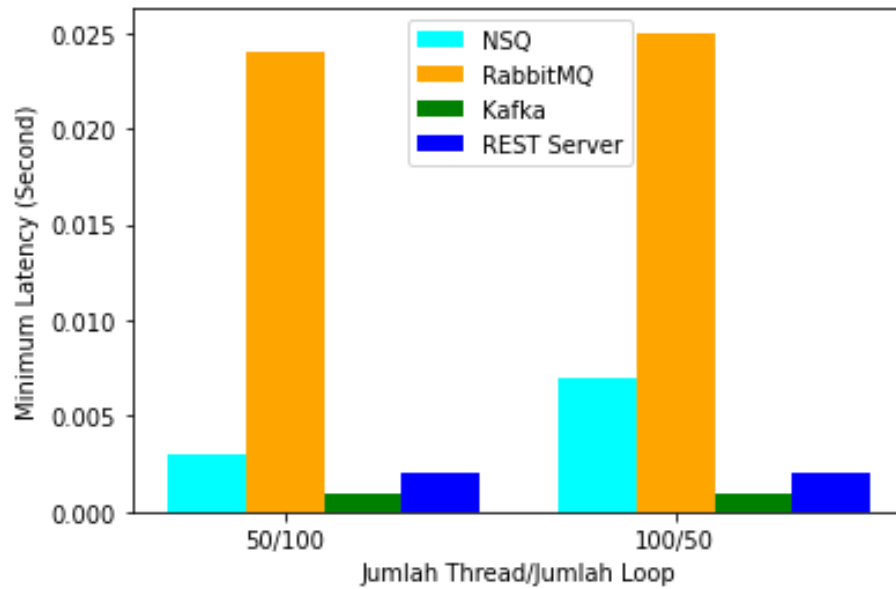
Maximum Latency*Memory Usage*

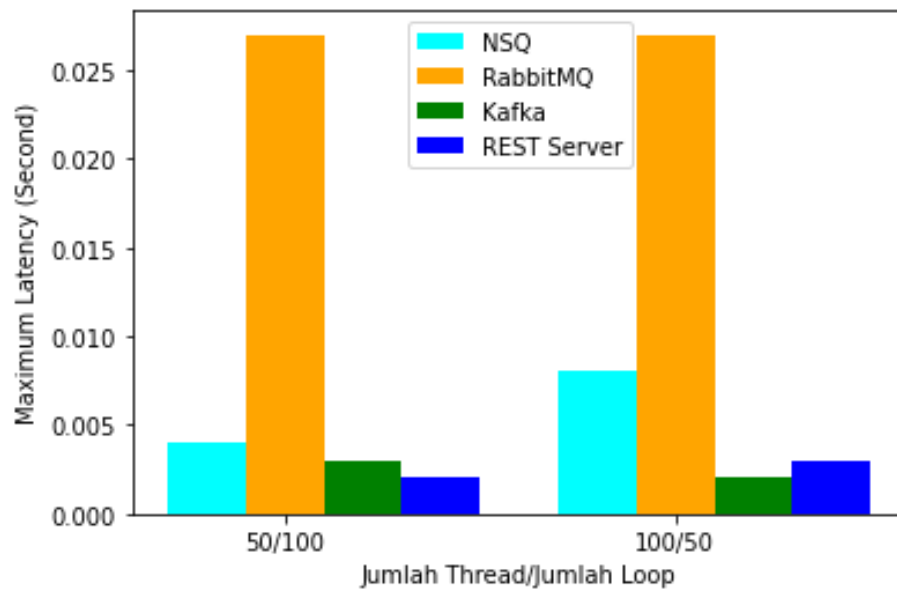
Lampiran 3. Hasil Pengujian Skenario Multiple Thread, Multiple Loop

Throughput



Minimum Latency



Maximum Latency*Memory Usage*