

**TUGAS AKHIR – EF234801**

# **PENGELOLAAN PENGGUNAAN INFRASTRUKTUR GPU UNTUK PENGGUNA BERBASIS DOCKER CONTAINER MENGUNAKAN JUPYTERLAB**

**Gloriyano Cristho Daniel Pepuho**  
NRP 5025201121

Dosen Pembimbing 1

**Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.**  
NIP 19810620 200501 1 003

Dosen Pembimbing 2

**Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.**  
NIP 19770824 200604 1 001

**Program Studi Strata 1 (S1) Teknik Informatika**

Departemen Teknik Informatika

Fakultas Teknologi Elektro dan Informatika Cerdas

Institut Teknologi Sepuluh Nopember

Surabaya

2025



**TUGAS AKHIR – EF234801**

**PENGELOLAAN PENGGUNAAN INFRASTRUKTUR  
GPU UNTUK PENGGUNA BERBASIS DOCKER  
CONTAINER MENGGUNAKAN JUPYTERLAB**

**Gloriyano Cristho Daniel Pepuho**

NRP 5025201121

Dosen Pembimbing 1

**Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.**

NIP 19810620 200501 1 003

Dosen Pembimbing 2

**Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.**

NIP 19770824 200604 1 001

**Program Studi Strata 1 (S1) Teknik Informatika**

Departemen Teknik Informatika

Fakultas Teknologi Elektro dan Informatika Cerdas

Institut Teknologi Sepuluh Nopember

Surabaya

2025

*[Halaman ini sengaja dikosongkan]*



**FINAL PROJECT - EF234801**

***MANAGING DISTRIBUTED GPU INFRASTRUCTURE  
USAGE FOR USERS BASED ON DOCKER  
CONTAINERS USING JUPYTERLAB***

**Gloriyano Cristho Daniel Pepuho**

NRP 5025201121

Advisor

**Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.**

NIP 19810620 200501 1 003

**Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.**

NIP 19770824 200604 1 001

**Undergraduate Study Program of Informatics**

Department of Informatics

Faculty of Intelligent Electrical and Informatics Technology

Institut Teknologi Sepuluh Nopember Surabaya

2025

*[Halaman ini sengaja dikosongkan]*

# LEMBAR PENGESAHAN

## PENGELOLAAN PENGGUNAAN INFRASTRUKTUR GPU UNTUK PENGGUNA BERBASIS DOCKER CONTAINER MENGGUNAKAN JUPYTERLAB

### TUGAS AKHIR

Diajukan untuk memenuhi salah satu syarat  
memperoleh gelar Sarjana Teknik pada  
Program Studi S-1 Teknik Informatika  
Departemen Departemen Teknik Informatika  
Fakultas Teknologi Elektro dan Informatika Cerdas  
Institut Teknologi Sepuluh Nopember

Oleh: **Gloriyano Cristho Daniel Pepuho**  
NRP. 5025201121

Disetujui oleh Tim Penguji Tugas Akhir:

Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.  
NIP: 19810620 200501 1 003

(Pembimbing I)

.....

Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.  
NIP: 19770824 200604 1 001

(Pembimbing II)

.....

Baskoro Adi Pratomo, S.Kom., M.Kom., Ph.D.  
NIP: 19870218 201404 1 001

(Penguji I)

.....

Prof. Tohari Ahmad, S.Kom., M.IT., Ph.D.  
NIP: 19750525 200312 1 002

(Penguji II)

.....

**SURABAYA**  
**Juli, 2025**

*[Halaman ini sengaja dikosongkan]*

# APPROVAL SHEET

## ***MANAGING DISTRIBUTED GPU INFRASTRUCTURE USAGE FOR USERS BASED ON DOCKER CONTAINERS USING JUPYTERLAB***

### **FINAL PROJECT**

Submitted to fulfill one of the requirements  
for obtaining a degree Bachelor of Engineering at  
Undergraduate Study Program of Informatics  
Department of Informatics  
Faculty of Intelligent Electrical and Informatics Technology  
Sepuluh Nopember Institute of Technology

By: **Gloriyano Cristho Daniel Pepuho**  
NRP. 5025201121

Approved by Final Project Examiner Team:

Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.  
NIP: 19810620 200501 1 003

(Advisor I)

.....

Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.  
NIP: 19770824 200604 1 001

(Co-Advisor II)

.....

Baskoro Adi Pratomo, S.Kom., M.Kom., Ph.D.  
NIP: 19870218 201404 1 001

(Examiner I)

.....

Prof. Tohari Ahmad, S.Kom., M.IT., Ph.D.  
NIP: 19750525 200312 1 002

(Examiner II)

.....

**SURABAYA**  
**July, 2025**



*[Halaman ini sengaja dikosongkan]*

## PERNYATAAN ORISINALITAS

Yang bertanda tangan dibawah ini:

Nama Mahasiswa / NRP : Gloriyano Cristho Daniel Pepuho / 5025201121  
Departemen : Departemen Teknik Informatika  
Dosen Pembimbing / NIP : Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D. /  
19810620 200501 1 003

Dengan ini menyatakan bahwa Tugas Akhir dengan judul "PENGELOLAAN PENGGUNAAN INFRASTRUKTUR GPU UNTUK PENGGUNA BERBASIS DOCKER CONTAINER MENGGUNAKAN JUPYTERLAB" adalah hasil karya sendiri, berfsifat orisinal, dan ditulis dengan mengikuti kaidah penulisan ilmiah.

Bilamana di kemudian hari ditemukan ketidaksesuaian dengan pernyataan ini, maka saya bersedia menerima sanksi sesuai dengan ketentuan yang berlaku di Institut Teknologi Sepuluh Nopember.

Surabaya, July 2025

Mengetahui  
Dosen Pembimbing

Mahasiswa

Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.  
NIP. 19810620 200501 1 003

Gloriyano Cristho Daniel Pepuho  
NRP. 5025201121

*[Halaman ini sengaja dikosongkan]*

## STATEMENT OF ORIGINALITY

The undersigned below:

Name of student / NRP : Gloriyano Cristho Daniel Pepuho / 5025201121  
Department : Informatics  
Advisor / NIP : Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D. /  
19810620 200501 1 003

Hereby declared that the Final Project with the title of "*MANAGING DISTRIBUTED GPU INFRASTRUCTURE USAGE FOR USERS BASED ON DOCKER CONTAINERS USING JUPYTERLAB*" is the result of my own work, is original, and is written by following the rules of scientific writing.

If in future there is a discrepancy with this statement, then I am willing to accept sanctions in accordance with provisions that apply at Sepuluh Nopember Institute of Technology.

Surabaya, July 2025

Acknowledged  
Advisor

Student

Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.      Gloriyano Cristho Daniel Pepuho  
NIP. 19810620 200501 1 003      NRP. 5025201121

*[Halaman ini sengaja dikosongkan]*

## ABSTRAK

Nama Mahasiswa : Gloriyano Cristho Daniel Pepuho  
Judul Tugas Akhir : PENGELOLAAN PENGGUNAAN INFRASTRUKTUR GPU UNTUK PENGGUNA BERBASIS DOCKER CONTAINER MENGGUNAKAN JUPYTERLAB  
Pembimbing : 1. Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.  
2. Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Penggunaan JupyterLab sebagai antarmuka interaktif untuk komputasi ilmiah dan pembelajaran semakin meluas. Namun, dalam lingkungan dengan banyak pengguna dan keterbatasan sumber daya, diperlukan sistem yang mampu mendistribusikan beban secara efisien. Penelitian ini bertujuan untuk membangun arsitektur JupyterHub yang terintegrasi dengan *service discovery*, sehingga setiap *instance* JupyterLab dapat dijalankan pada node yang memiliki sumber daya paling sesuai.

Sistem dikembangkan menggunakan pendekatan *container* berbasis Docker, dengan modifikasi pada *Spawner* untuk mendukung pemilihan node secara dinamis melalui Discovery API. Setiap node memanfaatkan Agent Service untuk mengirimkan informasi penggunaan sumber daya (CPU, RAM, GPU) secara periodik ke Redis dan PostgreSQL. Proses pemilihan node mempertimbangkan kapasitas dan beban terkini, dan dilakukan secara otomatis saat pengguna melakukan spawn JupyterLab.

Pengujian dilakukan dengan mensimulasikan pengguna secara bersamaan menggunakan. Hasil uji coba awal dengan 10 pengguna menunjukkan bahwa sistem mampu menangani permintaan secara serentak. Namun, pengujian lanjutan dalam skala besar masih diperlukan untuk mengevaluasi ketahanan sistem secara menyeluruh.

**Kata Kunci:** *Klaster GPU, Docker Container, JupyterLab, Pengelolaan pengguna*

*[Halaman ini sengaja dikosongkan]*

## ABSTRACT

*Name* : Gloriyano Cristho Daniel Pepuho  
*Title* : *MANAGING DISTRIBUTED GPU INFRASTRUCTURE USAGE FOR USERS  
BASED ON DOCKER CONTAINERS USING JUPYTERLAB*  
*Advisors* : 1. Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.  
2. Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

JupyterLab is widely used as an interactive interface for scientific computing and education. However, in multi-user environments with limited resources, a system capable of distributing workloads efficiently is required. This study aims to develop a JupyterHub architecture integrated with a service discovery mechanism, allowing each JupyterLab instance to be automatically deployed on the most suitable node.

The system is built using a container-based approach with Docker, and the Spawner component is modified to support dynamic node selection through a Discovery API. Each node runs an Agent Service that periodically sends resource usage information (CPU, RAM, GPU) to Redis and PostgreSQL. Node selection considers the current load and capacity, and is triggered automatically during the JupyterLab spawning process.

Testing was conducted by simulating concurrent users. Initial results with 10 users showed that the system was able to handle simultaneous requests without major issues. However, further testing on a larger scale is needed to fully evaluate the system's robustness.

***Keywords:*** *GPU Cluster, Docker Container, JupyterLab, User Management*



*[Halaman ini sengaja dikosongkan]*

## KATA PENGANTAR

Puji dan syukur kehadiran Tuhan Yang Maha Esa yang memberikan karunia, rahmat, dan pertolongan sehingga penulis dapat menyelesaikan penelitian tugas akhir yang berjudul 'PENGELOLAAN PENGGUNAAN INFRASTRUKTUR GPU UNTUK PENGGUNA BERBASIS DOCKER CONTAINER MENGGUNAKAN JUPYTERLAB'. Melalui kata pengantar ini, penulis mengucapkan terima kasih sebesar-besarnya kepada seluruh pihak yang telah membantu dan mendukung penulis selama mengerjakan penelitian tugas akhir ini, diantaranya adalah:

1. Tuhan Yang Maha Esa, atas karunia dan rahmat-Nya sehingga penulis dapat mencapai titik akhir perkuliahan strata satu di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember.
2. Kedua orang tua yang telah mendukung penulis selama berkuliah di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember.
3. Bapak Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D. dan Bapak Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D. sebagai dosen pembimbing yang telah membimbing, memberi arahan, dan masukan kepada penulis selama mengerjakan tugas akhir ini.
4. Dosen dan tenaga pendidik di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember yang telah memberikan pengetahuan, wawasan, dan pengalaman yang sangat berarti selama masa studi.
5. Pihak-pihak lain yang tidak dapat disebutkan satu persatu yang telah membantu penulis dalam pelaksanaan penelitian tugas akhir ini.

Akhir kata, semoga penelitian tugas akhir ini dapat memberikan kontribusi yang bermanfaat. Terima kasih dan permohonan maaf atas kekurangan dan kesalahan dalam pelaksanaan tugas akhir ini.

Surabaya, Juli 2025

Gloriyano Cristho Daniel Pepuho

*[Halaman ini sengaja dikosongkan]*

# DAFTAR ISI

<b>ABSTRAK</b>	<b>i</b>
<b>ABSTRACT</b>	<b>iii</b>
<b>KATA PENGANTAR</b>	<b>v</b>
<b>DAFTAR ISI</b>	<b>vii</b>
<b>DAFTAR GAMBAR</b>	<b>ix</b>
<b>DAFTAR TABEL</b>	<b>xi</b>
<b>DAFTAR KODE SUMBER</b>	<b>xiii</b>
<b>DAFTAR SINGKATAN</b>	<b>xv</b>
<b>1 PENDAHULUAN</b>	<b>1</b>
1.1 Latar Belakang . . . . .	1
1.2 Rumusan Masalah . . . . .	2
1.3 Batasan Masalah atau Ruang Lingkup . . . . .	2
1.4 Tujuan . . . . .	2
1.5 Manfaat . . . . .	2
1.6 Sistematika Penulisan . . . . .	3
<b>2 TINJAUAN PUSTAKA</b>	<b>5</b>
2.1 Hasil penelitian/perancangan terdahulu . . . . .	5
2.1.1 Containerisation for High Performance Computing Systems: Survey and Prospects . . . . .	5
2.1.2 An accessible infrastructure for artificial intelligence using a Docker- based JupyterLab in Galaxy . . . . .	5
2.2 Teori/Konsep Dasar . . . . .	6
2.2.1 Klaster GPU . . . . .	6
2.2.2 Docker . . . . .	6
2.2.3 JupyterLab . . . . .	8

2.2.4	JupyterHub . . . . .	9
2.2.5	Jupyter Enterprise Gateway . . . . .	11
2.2.6	Service Discovery . . . . .	14
2.2.7	Flask . . . . .	14
2.2.8	Redis . . . . .	15
2.2.9	PostgreSQL . . . . .	15
<b>3</b>	<b>METODOLOGI</b>	<b>17</b>
3.1	Metode yang Dirancang . . . . .	17
3.1.1	Service Discovery . . . . .	18
3.1.2	Service Agent . . . . .	20
3.1.3	JupyterHub . . . . .	20
3.1.4	Jupyter Enterprise Gateway . . . . .	21
3.2	Implementasi Sistem . . . . .	22
3.2.1	Implementasi Service Discovery . . . . .	22
3.2.2	Implementasi Service Agent . . . . .	31
3.2.3	Implementasi JupyterHub dan Jupyter Enterprise Gateway . . . . .	35
3.3	Peralatan Pendukung . . . . .	43
<b>4</b>	<b>HASIL dan PEMBAHASAN</b>	<b>47</b>
4.1	Hasil Implementasi . . . . .	47
4.1.1	Implementasi Service Agent . . . . .	47
4.1.2	Implementasi Service Discovery . . . . .	48
4.1.3	Hasil Implementasi JupyterHub . . . . .	51
4.2	Hasil Uji Coba . . . . .	52
4.2.1	Menjalankan Single User-Single Node GPU . . . . .	52
4.2.2	Menjalankan Single User-Multi Nodes CPU . . . . .	56
4.2.3	Uji Coba Multi-User Concurrent . . . . .	57
<b>5</b>	<b>PENUTUP</b>	<b>61</b>
5.1	Kesimpulan . . . . .	61
5.2	Saran . . . . .	61
	<b>DAFTAR PUSTAKA</b>	<b>63</b>
	<b>BIOGRAFI PENULIS</b>	<b>65</b>

## DAFTAR GAMBAR

2.1	Arsitektur Docker (Sumber: S. D. Team, 2024)	7
2.2	Arsitektur JupyterHub (Sumber: J. Team, 2024)	9
2.3	Arsitektur Komunikasi JEG (Sumber: Jupyter Team, 2025)	13
3.1	Arsitektur Penelitian	18
3.2	Diagram alir service discovery	19
3.3	Diagram alir service discovery dalam menampilkan ketersediaan <i>node</i>	20
3.4	Alur komunikasi JEG dan JupyterHub	21
3.5	Log dari container agent ketika melakukan registrasi node	35
4.1	Informasi dari setiap node berhasil disimpan ke Redis	50
4.2	Informasi dari setiap node berhasil disimpan ke PostgreSQL di tabel <i>nodes</i>	50
4.3	Menjalankan JupyterLab UI	51
4.4	Log <i>spawn</i> JupyterLab UI	51
4.5	Proses Registrasi dan Login pada JupyterHub	53
4.6	User memilih profil dan environment sesuai kebutuhan pada halaman JupyterHub	53
4.7	Terdapat 1 remote kernel, sesuai dengan jumlah node yang dipilih sebelumnya	55
4.8	JEG menjalankan <i>remote kernel</i>	55
4.9	Interaksi yang dilakukan di Jupyter Kernel	56
4.10	Spawn Multi Nodes CPU	56
4.11	Multi Nodes Kernel	57
4.12	Opsi untuk memilih load simulasi menggunakan 10 <i>user</i>	58
4.13	Proses <i>load test user</i>	58
4.14	Durasi <i>load test</i> untuk menjalankan 10 <i>user</i>	58
4.15	U	59
4.16	T	59

*[Halaman ini sengaja dikosongkan]*

## DAFTAR TABEL

3.1	Struktur Direktori Discovery Service . . . . .	22
3.2	Kolom dan Tipe Data Model Node . . . . .	24
3.3	Kolom dan Tipe Data Model NodeSelection . . . . .	25
3.4	Kolom dan Tipe Data Model NodeMetric . . . . .	25
3.5	Contoh Isi File .env dari <i>Discovery Service</i> . . . . .	28
3.6	Struktur Direktori Proyek JupyterHub . . . . .	36
3.7	Spesifikasi Peralatan Pendukung . . . . .	44
3.8	Daftar Perangkat Lunak Pendukung . . . . .	45
4.1	Daftar Endpoint REST API pada Discovery Service . . . . .	48



*[Halaman ini sengaja dikosongkan]*

## DAFTAR KODE SUMBER

3.1	Kode Semu untuk <code>app.py</code> . . . . .	23
3.2	Kode Semu untuk Model Node . . . . .	24
3.3	Kode Semu untuk Model NodeSelection . . . . .	24
3.4	Kode Semu untuk Model NodeMetric . . . . .	25
3.5	Kode Semu untuk Koneksi Redis Menggunakan ConnectionPool . . . . .	26
3.6	Kode Semu Penyimpanan Data Node ke Redis dengan TTL . . . . .	26
3.7	Kode Semu Fungsi Perhitungan Skor Node . . . . .	26
3.8	Kode Semu untuk <code>config.py</code> . . . . .	27
3.9	Kode Semu Proses Pembangunan Image Docker . . . . .	28
3.10	Kode Semu untuk <code>docker-compose.yml</code> . . . . .	29
3.11	Menjalankan Discovery Service via Docker Compose . . . . .	30
3.12	Kode Semu Fungsi Register Agent . . . . .	31
3.13	Kode Semu Kumpulan Informasi Sistem oleh Agent . . . . .	31
3.14	Deteksi Container JupyterLab . . . . .	32
3.15	Deteksi GPU Menggunakan <code>gpustat</code> . . . . .	32
3.16	Kode Semu Loop Registrasi Agent Tiap 15 Detik . . . . .	32
3.17	Langkah Konfigurasi Docker untuk Akses Remote . . . . .	33
3.18	Dockerfile untuk Membangun Agent Service . . . . .	33
3.19	Perintah untuk Build dan Menjalankan Agent . . . . .	34
3.20	Perintah untuk Push Image Agent . . . . .	34
3.21	Perintah untuk Menjalankan Agent di setiap Node . . . . .	34
3.22	Kode Semu file Jupyterhub Config . . . . .	36
3.23	Kode Semu untuk Konfigurasi Autentikasi . . . . .	37
3.24	Kode Semu untuk Konfigurasi Inti Hub . . . . .	37
3.25	Kode Semu untuk Konfigurasi Proxy . . . . .	38
3.26	Kode Semu untuk Logika MultiNodeSpawner . . . . .	38
3.27	Kode Semu untuk Generasi Konfigurasi Kernel JEG . . . . .	39
3.28	Kode Semu untuk Properti URL pada Spawner . . . . .	40
3.29	File Konfigurasi JEG . . . . .	40
3.30	Dockerfile JupyterHub . . . . .	41
3.31	Dockerfile JEG . . . . .	42

3.32 Orkestrasi beberapa layanan dengan Docker Compose . . . . .	42
4.1 Contoh log registrasi node . . . . .	47
4.2 Contoh respons Discovery API pada endpoint /available-nodes . . . . .	49
4.3 Contoh KernelSpec . . . . .	51

## DAFTAR SINGKATAN

<b>API</b>	Application Programming Interface
<b>CPU</b>	Central Processing Unit
<b>CLI</b>	Command Line Interface
<b>GPU</b>	Graphics Processing Unit
<b>GUI</b>	Graphical User Interface
<b>JEG</b>	Jupyter Enterprise Gateway
<b>NoSQL</b>	Not only Structured Query Language
<b>PAAS</b>	Platform as a Service
<b>SQL</b>	Structured Query Language

*[Halaman ini sengaja dikosongkan]*

# BAB I

## PENDAHULUAN

### 1.1 Latar Belakang

Peningkatan kebutuhan komputasi untuk machine learning (ML) dan kecerdasan buatan (AI) telah mendorong penggunaan Graphics Processing Unit (GPU) secara masif. Urgensi pergeseran ini ditegaskan oleh CEO NVIDIA, Jensen Huang, ketika menjadi pembicara utama di NVIDIA GPU Technology Conference (GTC) 2023, di mana Ia menyatakan, "**The starting point of the new world is a new type of computer, a new computing model... This is the accelerated computing model.**" Kutipan ini menggaris bawahi bahwa era komputasi modern menuntut arsitektur baru di mana GPU menjadi elemen krusial karena kemampuannya dalam pemrosesan paralel yang cepat dan efisien terutama untuk *training model deep learning*. Namun, seiring meningkatnya kebutuhan ini, muncul tantangan baru dalam pengelolaannya.

Salah satu tantangan utama adalah tingginya biaya atau *cost* dalam menginvestasi dan pemeliharaan infrastruktur GPU. Bagi banyak institusi, terutama di lingkungan pendidikan dan riset, pengadaan GPU dalam jumlah besar seringkali tidak memungkinkan. Di sisi lain, tidak jarang komputer dengan GPU yang sudah ada baik di laboratorium atau milik perorangan tidak dimanfaatkan secara optimal dan seringkali berada dalam keadaan *idle*.

Untuk menjawab tantangan tersebut, pengelolaan sumber daya dalam lingkungan multi-pengguna menjadi sangat penting. Pengelolaan yang tidak efisien dapat menyebabkan alokasi sumber daya yang tidak merata dan penurunan kinerja sistem. Diperlukan sebuah mekanisme alokasi dan penjadwalan sumber daya (*resource allocation and scheduling*) yang dinamis, yang dapat memastikan bahwa setiap pengguna mendapatkan akses yang adil dan efisien. Pendekatan penjadwalan tradisional seringkali tidak cukup untuk menangani karakteristik unik dari beban kerja AI

Pengelolaan sumber daya dalam lingkungan multi-pengguna menjadi sangat penting. Pengelolaan yang tidak efisien menyebabkan alokasi sumber daya tidak merata dan penurunan kinerja sistem. Diperlukan mekanisme alokasi dan penjadwalan sumber daya (*resource allocation and scheduling*) yang dinamis untuk memastikan setiap pengguna mendapatkan akses yang adil dan efisien.

Penelitian ini mengembangkan sistem pengelolaan infrastruktur GPU atau non-GPU terdistribusi untuk banyak pengguna. Sistem memanfaatkan teknologi kontainerisasi Docker untuk menciptakan lingkungan kerja yang terisolasi dan konsisten. JupyterLab digunakan sebagai antarmuka utama yang interaktif dan mudah diakses, memungkinkan pengguna menjalankan tugas komputasi tanpa konfigurasi yang rumit.

## 1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam tugas akhir ini adalah sebagai berikut:

1. Bagaimana merancang arsitektur yang mengintegrasikan JupyterHub dan *service discovery* agar setiap *instance* JupyterLab dapat memanfaatkan sumber daya komputasi (GPU dan CPU) yang tersebar di berbagai *node*?
2. Bagaimana sistem menangani dan mendistribusikan antrian permintaan dari banyak pengguna secara bersamaan (*concurrent requests*), dan bagaimana pengaruhnya terhadap alokasi sumber daya pada setiap *node*?

## 1.3 Batasan Masalah atau Ruang Lingkup

Batasan dalam pengerjaan tugas akhir ini adalah sebagai berikut:

1. Infrastruktur yang digunakan adalah komputer yang tersedia di lingkungan laboratorium atau institusi pendidikan, bukan *cloud platform* berskala besar
2. Implementasi sistem berfokus pada integrasi JupyterHub dengan Docker, serta pengembangan mekanisme *service discovery* untuk penjadwalan dinamis. Arsitektur ini tidak mencakup perbandingan mendalam dengan orkestrator lain seperti Docker Swarm atau Kubernetes.
3. Algoritma penjadwalan (*scheduling*) yang dikembangkan berfokus pada implementasi satu model fungsional (misalnya, berbasis skor beban atau round-robin) dan tidak melakukan analisis komparatif terhadap semua kemungkinan algoritma penjadwalan.

## 1.4 Tujuan

Tujuan dari pembuatan Tugas Akhir ini adalah sebagai berikut:

1. Merancang dan mengimplementasikan sebuah mekanisme penjadwalan sumber daya (*resource scheduling*) dinamis yang dapat mengalokasikan kontainer Docker ke *node* dengan beban paling optimal dalam kluster.
2. Membangun arsitektur sistem yang mengintegrasikan JupyterHub dengan *service discovery*, sehingga setiap pengguna dapat secara mudah memperoleh lingkungan kerja (*instance* JupyterLab) yang berjalan di atas sumber daya terdistribusi.
3. Menganalisis dan mengevaluasi kinerja sistem dalam skenario multi-pengguna, terutama dalam hal manajemen antrian, waktu eksekusi tugas, dan efisiensi alokasi sumber daya.

## 1.5 Manfaat

Manfaat dari penelitian ini adalah sebagai berikut:

1. Sistem ini dapat meningkatkan efisiensi pemanfaatan infrastruktur GPU yang terbatas, mendukung penelitian, dan pengembangan berbasis komputasi AI.

2. Memberikan akses yang mudah, terstruktur, dan terisolasi ke sumber daya GPU atau non-GPU tanpa memerlukan pengetahuan teknis mendalam tentang manajemen *server* atau Docker.

## 1.6 Sistematika Penulisan

Laporan penelitian tugas akhir ini terbagi menjadi:

1. **BAB I Pendahuluan**

Bab ini berisi latar belakang penelitian yang menjelaskan pentingnya pengelolaan infrastruktur GPU terdistribusi. rumusan masalah yang dihadapi dalam penggunaan GPU multi-user, batasan masalah dan ruang lingkup penelitian, tujuan yang ingin dicapai, manfaat penelitian, serta sistematika penulisan laporan.

2. **BAB II Tinjauan Pustaka**

Bab ini berisi tinjauan terhadap penelitian-penelitian terdahulu yang relevan dengan topik penelitian, teori dan konsep dasar yang meliputi klaster GPU, teknologi Docker, penjadwalan GPU, JupyterLab, dan JupyterHub. Bab ini menjadi landasan teoritis untuk melakukan pengembangan sistem.

3. **BAB III Metodologi**

Bab ini berisi perancangan arsitektur sistem yang mencakup service discovery, integrasi JupyterHub dan integrasi Jupyterhub dengan Jupyter Enterprise Gateway. Selain itu bab ini membahas peralatan apa saja yang digunakan pada saat penelitian serta setiap detail implementasi komponen yang dikembangkan.

4. **BAB IV Pengujian dan Analisa**

Bab ini berisi bab ini dirancang untuk memvalidasi fungsionalitas sistem, evaluasi performa dalam berbagai kondisi beban, analisis efisiensi penggunaan resource, serta pembahasan hasil pengujian terhadap tujuan penelitian yang telah ditetapkan

5. **BAB V Penutup**

Bab ini berisi kesimpulan dari penelitian yang merangkum pencapaian tujuan penelitian, kontribusi yang diberikan, serta saran untuk pengembangan dan penelitian lebih lanjut yang dapat dilakukan berdasarkan penelitian ini.



*[Halaman ini sengaja dikosongkan]*

## **BAB II**

### **TINJAUAN PUSTAKA**

#### **2.1 Hasil penelitian/perancangan terdahulu**

Dalam melakukan penelitian ini, penulis akan menggunakan beberapa penelitian terdahulu sebagai pedoman dan referensi dalam mengerjakan tugas akhir ini.

##### **2.1.1 Containerisation for High Performance Computing Systems: Survey and Prospects**

Pada artikel ini, peneliti melakukan survei tentang penggunaan *container* dalam sistem *High Performance Computing (HPC)*. Fokus utama adalah bagaimana *container*, seperti Docker, dapat meningkatkan portabilitas, efisiensi, dan isolasi lingkungan di *HPC*. Artikel ini juga mengkaji kelebihan *container* dibandingkan *virtual machine*, termasuk *overhead* yang lebih rendah dan waktu *startup* yang lebih cepat. Survei ini dilakukan dengan mengkaji literatur dari berbagai penelitian terbaru tentang penggunaan *container* di sistem *HPC*, termasuk studi kasus implementasi *container* dalam kluster GPU dan simulasi berbasis AI.

Hasil survei menunjukkan bahwa *container* memungkinkan *workload HPC* dijalankan di berbagai platform dengan efisiensi yang tinggi, menjadikannya solusi populer untuk komputasi terdistribusi. Selain itu, peneliti juga mengidentifikasi tantangan seperti integrasi dengan sistem manajemen kluster dan penjadwalan yang optimal. Kontribusi utama dari artikel ini adalah menyajikan analisis perbandingan yang mendalam antara *container* dan *virtual machine* dalam konteks *HPC*, serta menyarankan pendekatan penjadwalan yang lebih optimal untuk *container* di lingkungan *multi-user*.

Temuan ini relevan dengan penelitian ini, terutama dalam konteks penggunaan Docker untuk mengelola pengguna pada kluster GPU terdistribusi. Konsep efisiensi yang diangkat dalam artikel ini memberikan dasar teoritis untuk pengembangan mekanisme penjadwalan GPU yang akan digunakan dalam penelitian ini, terutama dalam hal mengurangi *overhead* dan memastikan alokasi sumber daya yang adil. Selain itu, contoh aplikasi *container* di sistem *HPC* yang disebutkan dalam artikel ini memberikan inspirasi untuk implementasi praktis dalam pengelolaan lingkungan kerja berbasis *container* (Zhou et al., 2022).

##### **2.1.2 An accessible infrastructure for artificial intelligence using a Docker-based JupyterLab in Galaxy**

Pada artikel ini, peneliti mengembangkan infrastruktur yang dapat diakses untuk kecerdasan buatan dengan memanfaatkan JupyterLab berbasis Docker di dalam platform *Galaxy*. Infrastruktur ini dirancang untuk mempermudah pengguna dalam mengakses alat komputasi AI melalui antarmuka berbasis web.

Hasilnya menunjukkan bahwa pendekatan ini meningkatkan portabilitas dan aksesibilitas bagi pengguna. Penggunaan *container* Docker memungkinkan pengelolaan lingkungan komputasi yang konsisten dan meminimalkan konfigurasi manual. Artikel ini juga menyoroti man-

faat JupyterLab dalam menyediakan antarmuka yang intuitif bagi pengguna. Temuan ini relevan dengan penelitian ini, terutama dalam konteks penggunaan JupyterLab berbasis Docker untuk mengelola sumber daya komputasi GPU. Artikel ini memberikan wawasan tentang bagaimana layanan komputasi yang memiliki antar muka dan *container* dapat meningkatkan efisiensi dan aksesibilitas sistem (Kumar et al., 2023).

## 2.2 Teori/Konsep Dasar

### 2.2.1 Klaster GPU

Klaster GPU adalah kumpulan unit pemrosesan grafis (GPU) yang terhubung dalam satu sistem untuk mendukung komputasi paralel intensif. Klaster ini sering digunakan untuk mempercepat pemrosesan aplikasi dengan kebutuhan komputasi tinggi, seperti pelatihan model *deep learning* dan simulasi ilmiah. Efisiensi komputasi dicapai dengan membagi beban kerja antar GPU secara terdistribusi. Setiap GPU bekerja secara paralel untuk menyelesaikan bagian tertentu dari tugas besar, memungkinkan pengurangan waktu pemrosesan dan penggunaan sumber daya secara optimal. Manajemen sumber daya yang baik diperlukan agar alokasi beban kerja berjalan efisien dan terkoordinasi. (Shikai Wang and Shang, 2024).

### 2.2.2 Docker

Docker merupakan *tool open-source* yang mengotomatisasi proses penyebaran aplikasi ke dalam wadah (*container*). Docker dikembangkan oleh tim di Docker, Inc (sebelumnya dikenal sebagai dotCloud Inc), salah satu pelopor di pasar *Platform-as-a-Service* atau (PAAS), dan dirilis di bawah lisensi Apache 2.0. Apa yang membuat Docker istimewa? Docker menyediakan platform untuk penyebaran aplikasi yang dibangun di atas lingkungan eksekusi *container* yang tervirtualisasi. Teknologi ini dirancang untuk menghadirkan lingkungan yang ringan dan cepat bagi pengembangan serta eksekusi aplikasi, sekaligus menyederhanakan alur kerja distribusi kode—mulai dari perangkat pengembang, lingkungan pengujian, hingga tahap produksi. Dengan kemudahan yang ditawarkannya, Docker memungkinkan pengguna memulai hanya dengan host minimal yang memiliki kernel Linux yang kompatibel dan biner Docker (Turnbull, 2014).

Docker memiliki beberapa komponen penting, seperti berikut:

- **Docker Client**  
Docker Client adalah antarmuka utama yang digunakan oleh pengguna untuk berinteraksi dengan Docker. Melalui Docker Client, pengguna dapat mengirim perintah seperti membangun, mendistribusikan, dan menjalankan *container*. Perintah-perintah ini kemudian diteruskan ke Docker Daemon untuk diproses. Docker Client mendukung penggunaan antarmuka command line (CLI) yang intuitif, sehingga memudahkan pengelolaan infrastruktur container.
- **Docker Daemon**  
Docker Daemon adalah proses latar belakang yang bertanggung jawab untuk menangani perintah yang diterima dari Docker Client. Fungsinya meliputi pembuatan dan pengelolaan berbagai objek Docker, seperti *images*, *containers*, *networks*, dan *volumes*. Docker Daemon memastikan *container* berjalan dengan stabil dan memonitor aktivitasnya. Ia juga berperan penting dalam komunikasi dengan *registry* untuk *push* atau *pull* Docker images.
- **Docker Container**  
Docker Container adalah unit eksekusi yang ringan dan mandiri. *Container* ini berisi

semua komponen yang diperlukan untuk menjalankan aplikasi, termasuk kode aplikasi, pustaka, dependensi, dan konfigurasi. Karena sifatnya yang terisolasi, *container* memberikan lingkungan konsisten untuk aplikasi, terlepas dari perbedaan konfigurasi sistem di berbagai host.

- Docker Images

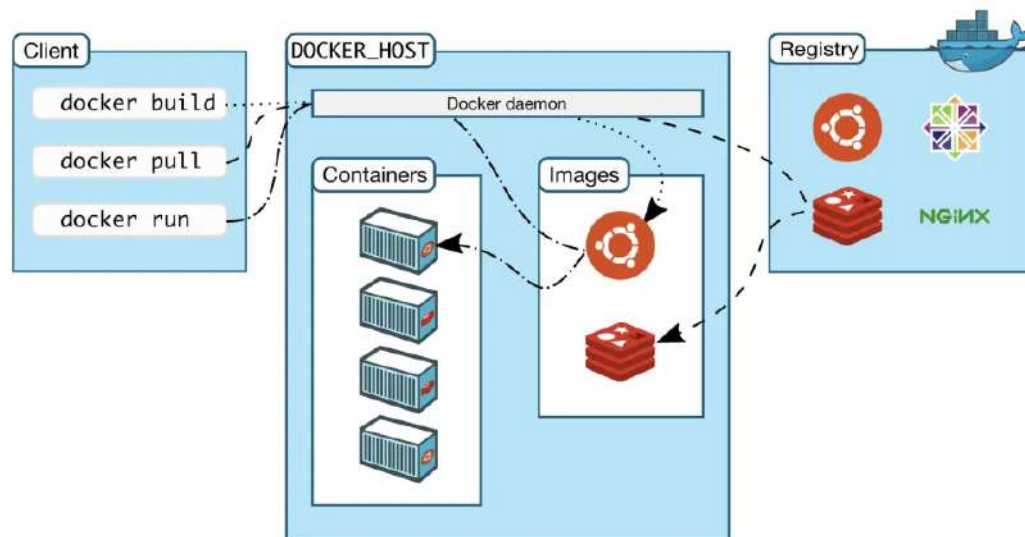
Docker Images adalah *template read-only* yang menjadi dasar untuk membangun Docker Container. Image ini mencakup semua dependensi, pustaka, dan file yang diperlukan untuk menjalankan aplikasi dalam container. Pengguna dapat membuat *images* dari *Dockerfile* atau *pull images* yang sudah ada dari Docker Hub atau *registry* lainnya. *Images* bersifat modular dan dapat diupdate atau digunakan kembali untuk berbagai kebutuhan.

- Registry

Registry adalah layanan penyimpanan dan distribusi Docker Images. Docker menyediakan *registry* publik seperti Docker Hub, tempat pengguna dapat mengunggah, menyimpan, dan berbagi *images*. Selain itu, pengguna juga dapat mengatur *registry* privat untuk kebutuhan spesifik organisasi. Registry mempermudah pengelolaan *images* dalam pengembangan kolaboratif dan siklus hidup *container*.

Arsitektur Docker dapat dilihat pada gambar 2.2. Dalam arsitektur ini, Docker Client berfungsi sebagai jembatan antara pengguna dan sistem Docker, di mana setiap perintah yang dikirimkan oleh pengguna akan diteruskan ke Docker Daemon yang berjalan pada sistem Docker Host.

Docker Daemon kemudian akan menjalankan proses yang dibutuhkan, mulai dari menarik *image* (*pull*) dari Docker Registry, membangun *container* dari *image* tersebut, hingga menjalankan dan mengelola siklus hidup *container*. Docker Registry sendiri berperan sebagai tempat penyimpanan dan distribusi Docker Image, baik melalui *registry* publik seperti Docker Hub, maupun *registry* privat yang disiapkan secara internal.



Gambar 2.1: Arsitektur Docker (Sumber: S. D. Team, 2024)

### 2.2.2.1 Docker Compose

Docker Compose adalah alat yang digunakan untuk mendefinisikan dan menjalankan aplikasi *multi-container* dalam lingkungan Docker. Dengan Docker Compose, pengguna dapat

menuliskan konfigurasi seluruh layanan yang dibutuhkan dalam satu berkas YAML (biasanya bernama ***docker-compose.yml***) (Docker Documentation, 2025).

Konsep utama dari Docker Compose adalah deklaratif. Setiap layanan, seperti aplikasi utama, database, cache, atau reverse proxy, dapat didefinisikan dalam satu file, lalu dijalankan secara bersamaan menggunakan satu perintah `docker compose up`. Ini menyederhanakan proses manajemen dan orkestrasi *container*, terutama pada lingkungan pengembangan atau pengujian.

Docker Compose mendukung berbagai fitur penting, seperti:

- Penentuan jaringan khusus untuk komunikasi antar *container*.
- Pendefinisian volume untuk penyimpanan data persisten atau permanen.
- Pengelolaan *multi-container* dalam 1 file YAML.

Dalam konteks penelitian ini, Docker Compose digunakan untuk menjalankan beberapa komponen layanan secara bersamaan, seperti JupyterHub, Redis, dan Discovery API, dalam satu lingkungan terisolasi namun saling terhubung. Hal ini memudahkan proses pengembangan, pengujian, dan pengelolaan sistem yang terdiri atas banyak layanan terpisah.

### 2.2.3 JupyterLab

JupyterLab adalah antarmuka pengguna berbasis web untuk Project Jupyter yang menyediakan lingkungan pengembangan interaktif yang fleksibel dan modular. JupyterLab memungkinkan pengguna untuk bekerja dengan *notebook*, file, *terminal*, dan editor teks dalam satu antarmuka terpadu yang dapat disesuaikan (J. D. Team, 2024).

JupyterLab berperan sebagai antarmuka utama yang memungkinkan pengguna mengakses sumber daya GPU secara interaktif. Setiap pengguna akan mendapatkan instance JupyterLab yang berjalan dalam Docker container, memberikan lingkungan kerja yang konsisten dan aman. Integrasi dengan Jupyter Enterprise Gateway (JEG) memungkinkan pengguna untuk menjalankan kernel notebook pada sumber daya kluster secara terdistribusi, langsung dari antarmuka JupyterLab.

JupyterLab sendiri dipilih karena kemudahan penggunaannya dalam lingkungan multi-pengguna dan kompatibilitasnya dengan Docker container, sehingga cocok untuk implementasi sistem penjadwalan GPU yang diusulkan dalam penelitian ini. Jupyterlab terdiri dari beberapa komponen utama yang bekerja secara sinergis, yaitu:

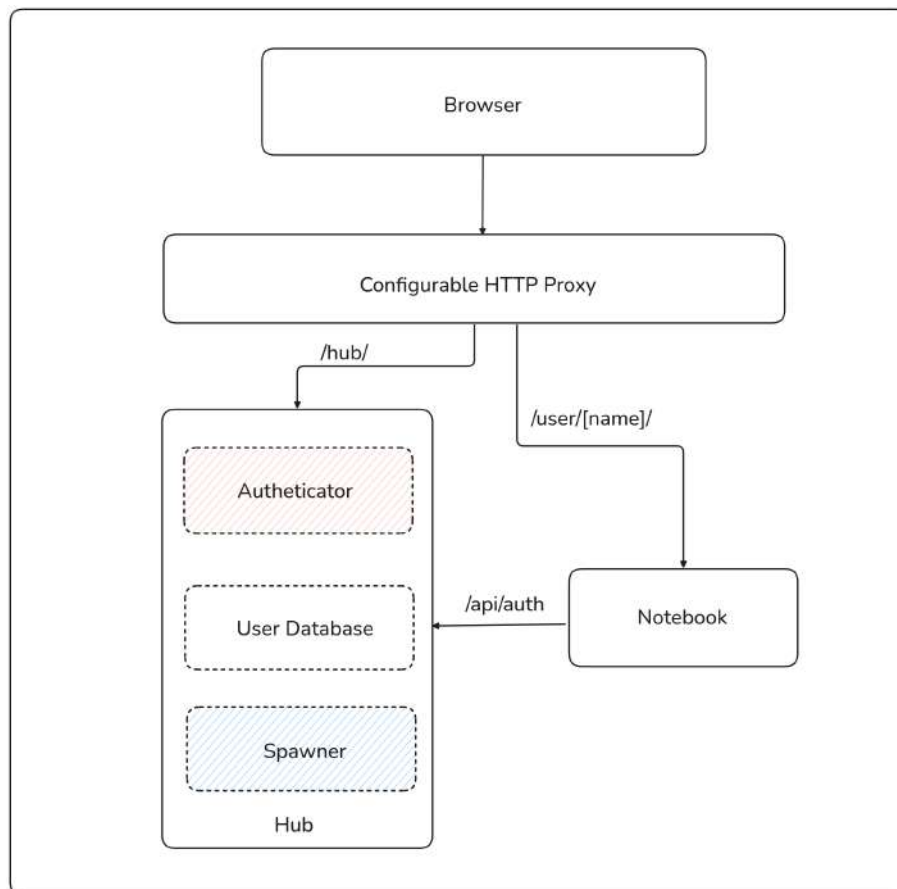
1. **Notebook Interface:** Menyediakan lingkungan interaktif untuk menjalankan kode Python, R, atau bahasa pemrograman lainnya dengan dukungan visualisasi data yang kaya.
2. **File Browser:** Memungkinkan navigasi dan manajemen file dalam sistem, termasuk upload dan download file secara langsung melalui antarmuka web.
3. **Terminal:** Akses terminal penuh yang terintegrasi dalam antarmuka web, memungkinkan eksekusi perintah sistem langsung dari browser.
4. **Extension System:** Mendukung plugin dan ekstensi untuk memperluas fungsionalitas sesuai kebutuhan spesifik pengguna.

## 2.2.4 JupyterHub

JupyterHub adalah platform *open-source* yang memungkinkan banyak pengguna untuk mengakses dan menjalankan lingkungan Jupyter Notebook secara terisolasi melalui antarmuka web. Dirancang untuk mendukung skenario multi-pengguna, JupyterHub sangat cocok digunakan dalam lingkungan pendidikan, penelitian, dan industri yang memerlukan akses bersama ke sumber daya komputasi (J. Team, 2024).

JupyterHub terdiri dari tiga komponen utama yang bekerja secara sinergis, yaitu:

1. **Hub:** Komponen inti yang bertanggung jawab atas manajemen akun pengguna, proses autentikasi, dan koordinasi peluncuran server notebook individu melalui mekanisme yang disebut Spawner.
2. **Proxy:** Berfungsi sebagai gerbang utama yang menerima semua permintaan HTTP dari pengguna dan meneruskannya ke Hub atau server notebook pengguna yang sesuai. Secara default, JupyterHub menggunakan configurable-http-proxy yang dibangun di atas node-http-proxy.
3. **Single-User Notebook:** Server Jupyter Notebook yang dijalankan secara terpisah untuk setiap pengguna setelah proses autentikasi berhasil. Server ini memungkinkan pengguna untuk menjalankan kode dan berinteraksi dengan lingkungan Jupyter secara pribadi.



Gambar 2.2: Arsitektur JupyterHub (Sumber: J. Team, 2024 )

## A. Alur Kerja JupyterHub

Proses interaksi pengguna dengan JupyterHub dapat dijelaskan sebagai berikut:

1. Akses Awal: Pengguna mengakses JupyterHub melalui web browser dengan mengunjungi alamat IP atau nama domain yang telah dikonfigurasi.
2. Autentikasi: Data login yang dimasukkan oleh pengguna dikirim ke komponen *Authenticator* untuk validasi. Jika valid, pengguna akan dikenali dan diizinkan untuk melanjutkan.
3. Peluncuran Server Notebook: Setelah autentikasi berhasil, JupyterHub akan meluncurkan *instance* Jupyter untuk pengguna tersebut menggunakan *spawner*.
4. Konfigurasi Proxy: Proxy dikonfigurasi untuk meneruskan permintaan dengan URL tertentu (misalnya, `/user/[username]`) ke server notebook pengguna yang sesuai.
5. Penggunaan Lingkungan Jupyter: Pengguna diarahkan ke server notebook pribadi mereka, di mana mereka dapat mulai bekerja di *instance* Jupyter.

## B. Melakukan kustomisasi dan menambah extension

JupyterHub dirancang dengan fleksibilitas tinggi, memungkinkan kustomisasi melalui dua komponen utama:

- **Authenticator:** Mengelola proses autentikasi pengguna. Jupyterhub mendukung berbagai metode autentikasi, termasuk:
  - **PAMAuthenticator:** Memungkinkan *user login* ke JupyterHub menggunakan *Linux user account*.
  - **OAuthAuthenticator:** Mendukung autentikasi menggunakan OAuth2, seperti Github, Google, atau GitLab.
  - **LDAPAuthenticator:** Memungkinkan *user login* menggunakan akun yang tersimpan di server LDAP (Lightweight Directory Access Protocol), seperti OpenLDAP atau Microsoft Active Directory.
  - **NativeAuthenticator:** Autentikator internal JupyterHub yang menyediakan halaman registrasi dan manajemen pengguna secara mandiri. Pada implementasi ini, *NativeAuthenticator* digunakan untuk menyederhanakan proses login dan pendaftaran pengguna secara terpusat tanpa perlu tanpa perlu menggunakan PAM atau integrasi eksternal seperti LDAP atau OAuth.
- **Spawner:** Mengontrol cara peluncuran *server notebook* untuk setiap pengguna. Beberapa jenis Spawner yang umum digunakan antara lain:
  - **DockerSpawner:** Menjalankan *server notebook* dalam *Docker container*, memberikan isolasi lingkungan yang lebih baik.
  - **KubeSpawner:** Menggunakan Kubernetes untuk mengelola dan menjalankan *server notebook* di lingkungan Kubernetes.

- **MultiNodeSpawner (custom spawner)**: Turunan dari **DockerSpawner** yang telah dimodifikasi untuk mendukung pemilihan node secara dinamis menggunakan *Service Discovery API*. Spawner ini memungkinkan peluncuran *container* JupyterLab pada node berbeda berdasarkan kapasitas sumber daya seperti RAM, CPU, GPU, serta skor load dari node. Pemilihan node dilakukan sebelum proses *spawn* dimulai, memastikan distribusi beban yang efisien dalam arsitektur *multi-server*.

Kemampuan untuk menyesuaikan dan memperluas JupyterHub melalui Authenticator dan Spawner memungkinkan integrasi yang mulus dengan berbagai infrastruktur dan kebutuhan spesifik pengguna.

### C. Kustomisasi Halaman Antarmuka

Selain kustomisasi fungsional, JupyterHub juga memungkinkan penyesuaian pada halaman antarmuka pengguna, terutama halaman opsi spawner (Server Options). Dalam penelitian ini, kustomisasi dilakukan dengan menyediakan template HTML dan JavaScript khusus yang ditempatkan dalam direktori **form/** pada struktur proyek.

Tujuan dari kustomisasi halaman ini adalah untuk menyediakan form atau halaman yang interaktif bagi pengguna, di mana mereka dapat memilih:

- Profil komputasi (misalnya, Single CPU, Single GPU, Multi CPU dan Multi GPU) sesuai kebutuhan.
- Environment (pemilihan Docker *image*) dan konfigurasi node (misalnya, memilih berapa jumlah node yang akan digunakan).

Proses ini memungkinkan pengguna untuk mengonfigurasi lingkungan kerja mereka sebelum server JupyterLab diluncurkan. Contoh tampilan dari halaman registrasi, login, dan halaman "Server Options" yang telah dikustomisasi dapat dilihat pada Gambar 4.6 di Bab 4 (Pengujian) menunjukkan proses pemilihan node secara keseluruhan dan *environment* secara rinci.

## 2.2.5 Jupyter Enterprise Gateway

Jupyter Enterprise Gateway (JEG) adalah *open-source* dari ekosistem Jupyter Project yang memungkinkan eksekusi *kernel* notebook secara terdistribusi pada kluster. JEG dirancang untuk skenario *multi-user* dan *multi-kernel* di lingkungan *enterprise*, di mana *kernel* dapat dijalankan pada resource yang berbeda seperti node CPU atau GPU dalam kluster Kubernetes atau Apache Yarn Hadoop(*platform* terdistribusi).

Dengan JEG, kernel tidak perlu dijalankan langsung pada *node* JupyterHub atau JupyterLab. Sebaliknya, JEG bertindak sebagai perantara (*gateway*) yang menerima permintaan eksekusi kernel dan mengarahkannya ke *node* yang memiliki *resource* sesuai kebutuhan pengguna. Hal ini meningkatkan skalabilitas dan memungkinkan penggunaan sumber daya yang lebih optimal dalam arsitektur komputasi terdistribusi (JEG Development Team, 2025).

### A. Fitur Utama JEG



1. Remote Kernel Execution: Memungkinkan *kernel notebook* berjalan di *remote node* tanpa perlu instalasi *kernel* di *node frontend*.
2. Multi-tenancy: Mendukung banyak pengguna untuk mengeksekusi *kernel* di *environment* yang telah disediakan.
3. Scalability: Dapat diintegrasikan dengan *cluster manager* (misal Kubernetes atau Apache Hadoop YARN) untuk men-deploy kernel secara elastis.

Dalam penelitian ini, JEG digunakan untuk menghubungkan JupyterHub dengan *kernel* yang dijalankan pada setiap *node* atau komputer, sehingga setiap pengguna dapat menjalankan kode mereka pada *node* yang tersedia tanpa konfigurasi manual yang kompleks di sisi client.

### 2.2.5.1 Arsitektur dan Process Proxy pada Jupyter Enterprise Gateway

Salah satu fitur penting dari Jupyter Enterprise Gateway (JEG) adalah kemampuannya untuk menjalankan kernel notebook secara terdistribusi, tanpa harus menjalankan kernel di mesin JupyterHub atau JupyterLab secara langsung. Ini dimungkinkan karena adanya komponen Process Proxy.

#### A. Komponen Arsitektur JEG

JEG bekerja dengan cara menerima permintaan peluncuran kernel dari client (JupyterHub), lalu mengarahkan permintaan tersebut ke host atau node lain menggunakan sebuah perantara bernama process proxy. Beberapa komponen utama JEG antara lain:

- **EnterpriseGatewayApp:** Komponen utama yang menerima permintaan kernel dari client.
- **KernelSpecManager:** Mengelola daftar kernel yang tersedia, dan menentukan konfigurasi eksekusinya.
- **RemoteKernelManager:** Mengelola lifecycle kernel yang dijalankan di remote node.
- **ProcessProxy:** Menentukan bagaimana kernel akan dijalankan, termasuk metode akses ke node (SSH, Docker, Kubernetes, dsb).

#### B. Konsep Process Proxy

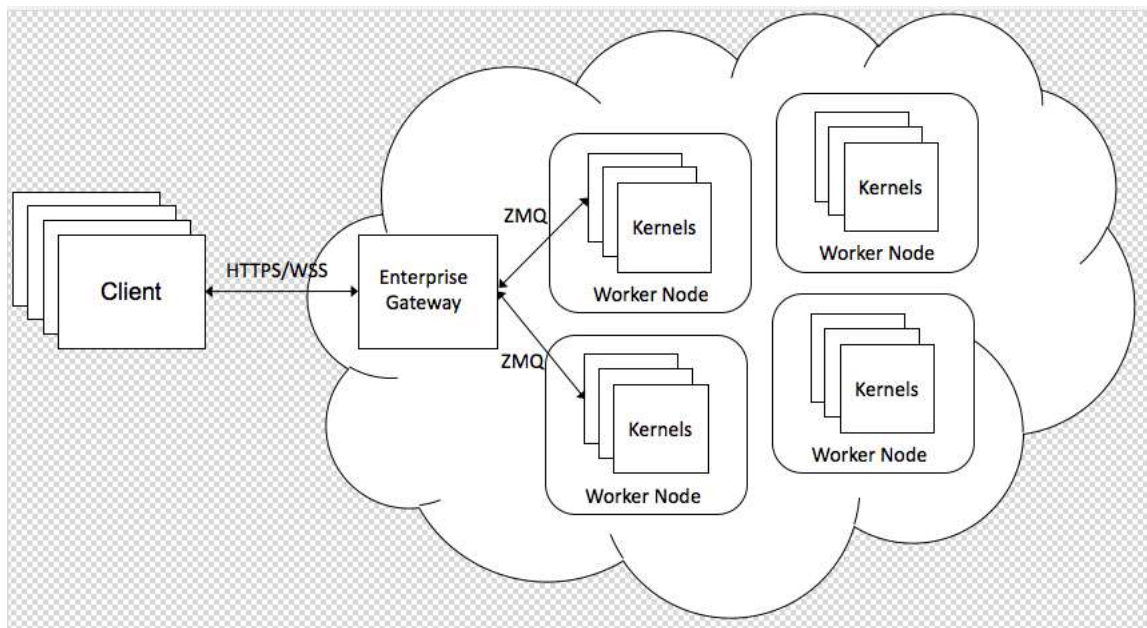
Process proxy adalah abstraksi yang memungkinkan JEG menjalankan kernel di luar host lokal. Beberapa jenis ProcessProxy yang umum digunakan:

- **LocalProcessProxy:** Menjalankan kernel di host lokal (default).
- **RemoteProcessProxy:** Menggunakan koneksi SSH ke host lain untuk menjalankan kernel.
- **DockerProcessProxy:** Menjalankan kernel di dalam container Docker.
- **KubernetesProcessProxy:** Mengelola kernel melalui Kubernetes pod.

Dalam implementasi penelitian ini, **RemoteProcessProxy** digunakan untuk meluncurkan kernel ke *node-node* yang telah dipilih oleh Discovery Service. Konfigurasi ini memungkinkan JEG menjalankan kernel di IP *node* tertentu yang ditentukan secara dinamis.

## B. Arsitektur Komunikasi Jupyter Enterprise Gateway

Gambar berikut menggambarkan arsitektur komunikasi client dengan Jupyter Enterprise Gateway (JEG) dan *node-node worker*:



Gambar 2.3: Arsitektur Komunikasi JEG (Sumber: Jupyter Team, 2025 )

- Client (JupyterLab/JupyterHub): Mengakses sistem melalui protokol HTTPS atau WSS untuk melakukan permintaan eksekusi *kernel*.
- Enterprise Gateway: Bertindak sebagai perantara yang menerima permintaan dari *client*, kemudian meneruskannya ke *worker node* yang ditentukan.
- Worker Node: Menjalankan proses eksekusi *kernel* secara aktual. Setiap *worker* dapat menjalankan satu atau lebih kernel secara paralel dalam *container*.

Komunikasi antara komponen-komponen ini memanfaatkan dua protokol penting, yaitu ZMQ dan WSS, yang berperan dalam menjaga koneksi real-time dan pertukaran data antar proses:

- **MQ (ZeroMQ):** Merupakan pustaka *messaging* yang digunakan untuk komunikasi antara JEG dan *kernel* yang berjalan di *node*. ZMQ menyediakan kanal interaktif berbasis *socket*, memungkinkan pertukaran pesan seperti eksekusi kode, *input/output stream*, dan kontrol *kernel* secara asinkron (ZMQ Authors, 2025).

- **WSS (WebSocket Secure):** Digunakan untuk menjaga koneksi dua arah yang persisten dan aman antara *client* (misalnya JupyterLab) dan Enterprise Gateway. WSS memungkinkan pengguna menerima hasil eksekusi kode secara langsung tanpa perlu melakukan *reload* halaman di *browser*, serta menjamin keamanan data melalui enkripsi selama sesi aktif (Heroku Team, 2025).

Komunikasi antara *client* dan Enterprise Gateway terjadi melalui protokol aman seperti HTTPS atau WSS (WebSocket Secure). Setelah permintaan diterima, JEG menggunakan ZeroMQ (ZMQ) untuk menginisialisasi dan mengelola koneksi ke *kernel* yang berjalan di *worker node* secara *remote*. Arsitektur ini memungkinkan pemisahan yang jelas antara komponen *frontend* (*client*) dan *backend* (*kernel*), serta mendukung skalabilitas tinggi dalam eksekusi paralel *kernel* (Jupyter Team, 2025).

Dengan pendekatan ini, setiap *kernel* dijalankan terisolasi di *nodes* terdistribusi yang berbeda sesuai dengan kondisi beban, serta memungkinkan sistem untuk menangani lebih banyak pengguna secara bersamaan tanpa harus membebani node pusat.

### 2.2.6 Service Discovery

Service Discovery adalah mekanisme inti dalam arsitektur sistem terdistribusi yang memungkinkan berbagai layanan (*services*) dan *node* untuk saling menemukan dan berkomunikasi secara otomatis tanpa perlu melakukan konfigurasi alamat IP dan port secara manual (hard-coding). Dalam lingkungan yang dinamis di mana *node* dapat bergabung atau meninggalkan kluster kapan saja, Service Discovery berfungsi sebagai sebuah registri (registry) terpusat yang berisi informasi terkini setiap *node*.

Dalam konteks penelitian ini, Service Discovery memegang peran krusial sebagai otak dari sistem penjadwalan. Komponen ini dirancang sebagai layanan pusat yang bertugas untuk menerima, menyimpan, dan menyediakan informasi status sumber daya dari setiap node yang tergabung dalam kluster GPU. Informasi yang dikumpulkan mencakup metrik vital seperti utilisasi CPU, RAM, ketersediaan dan penggunaan GPU, serta jumlah container yang sedang aktif di setiap node. Data tersebut dikirimkan secara periodik oleh Service Agent, sebuah komponen ringan yang berjalan di setiap node untuk memantau kondisi sistem secara lokal. Informasi ini kemudian diolah untuk mendukung pengambilan keputusan secara *real-time*. Ketika seorang pengguna meminta environment baru, JupyterHub akan berkomunikasi dengan Service Discovery untuk mendapatkan rekomendasi node mana yang memiliki beban paling ringan dan paling optimal untuk menjalankan container JupyterLab. Dengan demikian, mekanisme ini memastikan alokasi sumber daya yang efisien dan seimbang di seluruh kluster.

Service Discovery dalam penelitian ini berbentuk layanan API yang dibangun menggunakan *framework* Flask, Redis sebagai layanan penyimpanan yang *real-time* dan PostgreSQL sebagai basis data yang menyimpan data secara persisten.

### 2.2.7 Flask

Flask adalah *micro web-framework* berbasis Python yang digunakan untuk membangun aplikasi web dan REST API. Framework ini dikembangkan oleh Armin Ronacher dan pertama kali dirilis pada tahun 2010. Flask menawarkan fleksibilitas tinggi dan kemudahan penggunaan tanpa memaksakan struktur proyek tertentu, sehingga cocok untuk membangun layanan ringan seperti REST API yang digunakan dalam proyek ini (Flask Pallets Team, 2025).

Pada sistem ini, Flask digunakan untuk membangun Discovery Service, yaitu layanan yang

menerima dan menyediakan informasi status *real-time* dari seluruh node dalam kluster. Keunggulan Flask terletak pada kemampuannya dalam menangani *routing*, integrasi dengan berbagai ekstensi (seperti Flask-Migrate untuk database, dan Flask-CORS untuk akses lintas origin), serta dukungan terhadap pengembangan modular dengan menggunakan *library* atau pustaka *blueprint*.

Penggunaan Flask sebagai basis Discovery Service memungkinkan komunikasi antar layanan (seperti Agent dan JupyterHub) dilakukan melalui protokol HTTP.

### 2.2.8 Redis

Redis (Remote Dictionary Server) adalah sistem basis data NoSQL berbasis key-value yang berjalan di memori (*in-memory*). Redis mendukung berbagai struktur data seperti *string*, *hash*, *list*, *set*, dan *sorted set*. Karena berbasis memori, Redis menawarkan kecepatan baca-tulis yang sangat tinggi, sehingga sering digunakan dalam aplikasi *real-time*, *caching*, dan *message queue* (Redis Team, 2025).

Dalam penelitian ini, Redis digunakan sebagai penyimpanan sementara (*volatile*) untuk menyimpan status node yang dikirim oleh Agent setiap 15 detik. Data yang disimpan meliputi penggunaan CPU, RAM, status GPU, dan jumlah container aktif di setiap node. Redis juga dilengkapi dengan fitur *Time-To-Live (TTL)* yang digunakan untuk mendeteksi apakah suatu node masih aktif atau tidak. Dengan cara ini, Redis mendukung pengambilan keputusan secara *real-time* oleh Discovery Service ketika memilih node terbaik.

### 2.2.9 PostgreSQL

PostgreSQL adalah basis data relasional *open-source* yang dikenal karena stabilitas, standar SQL, serta dukungan fitur tingkat lanjut seperti transaksi ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*), indexing kompleks, dan extensibility (PostgreSQL Development Team, 2025).

Pada proyek ini, PostgreSQL digunakan sebagai penyimpanan persisten yang meliputi informasi node, profil pengguna, hasil seleksi node, dan riwayat metrik pemantauan. Berbeda dengan Redis yang menyimpan data *real-time*, PostgreSQL menyimpan data historis yang dibutuhkan untuk audit, analisis performa jangka panjang, dan manajemen sistem secara lebih komprehensif.

Dengan menggabungkan PostgreSQL dan Redis, sistem dapat memperoleh manfaat dari keduanya: kecepatan Redis untuk kebutuhan *real-time* dan keandalan PostgreSQL untuk data persisten.

*[Halaman ini sengaja dikosongkan]*

## BAB III

# METODOLOGI

Bab ini menjelaskan perancangan dan implementasi sistem pengelolaan sumber daya GPU secara terdistribusi menggunakan container Docker, JupyterHub dan Jupyter Enterprise Gateway. Sistem dirancang untuk mendukung penggunaan secara multi-pengguna dengan penjadwalan node berbasis beban kerja dan integrasi antarkomponen melalui Discovery Service.

Pembahasan pada bab ini meliputi arsitektur sistem secara keseluruhan, implementasi masing-masing komponen utama, serta perangkat lunak pendukung yang digunakan selama proses pengembangan.

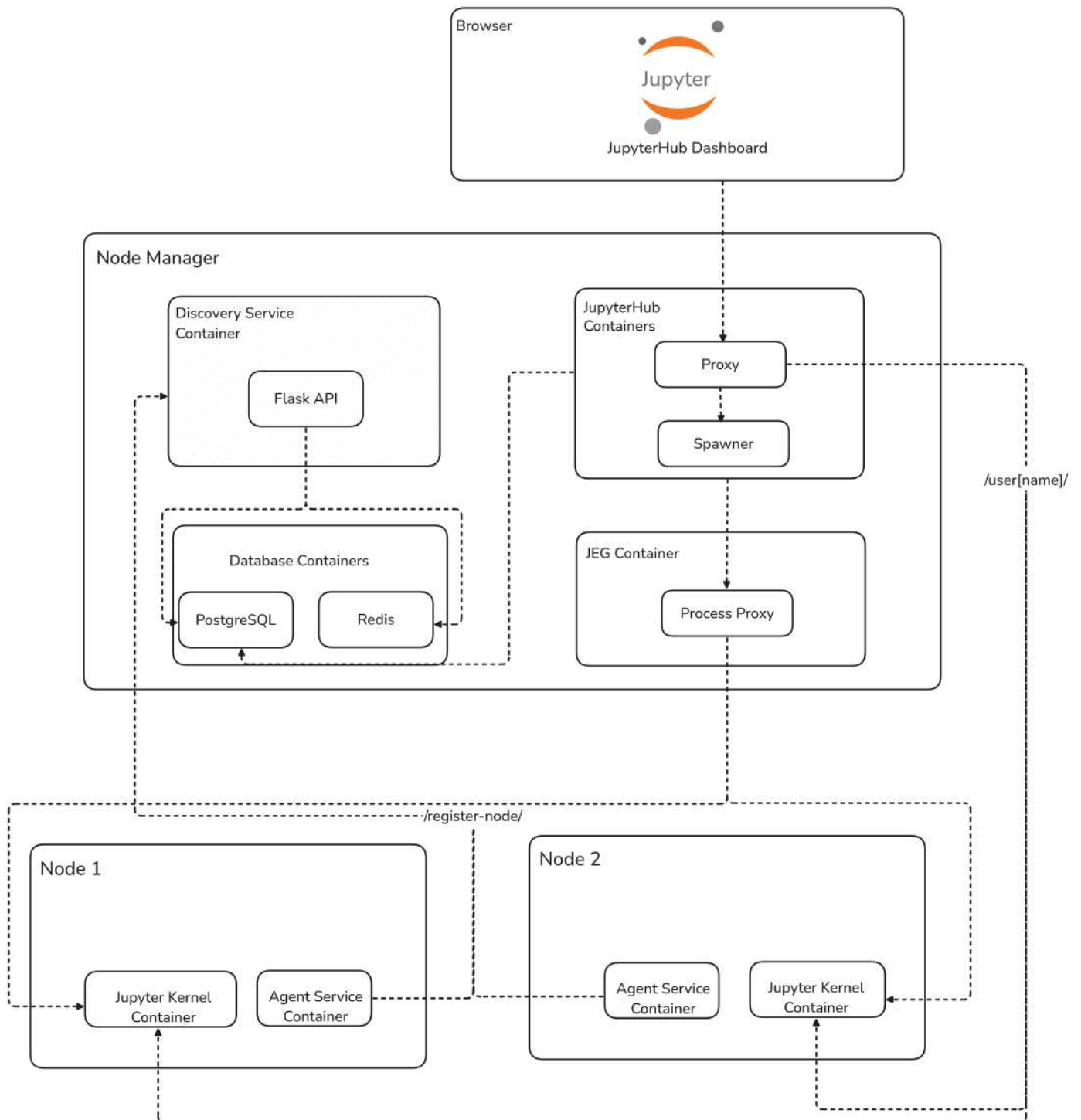
### 3.1 Metode yang Dirancang

Penelitian ini diawali dengan proses perancangan sistem yang bertujuan untuk memungkinkan pengelolaan sumber daya baik itu GPU maupun non-GPU secara efisien dan adil bagi banyak pengguna. Sistem dikembangkan untuk dapat berjalan dalam lingkungan terdistribusi dengan infrastruktur multi-node berbasis *container* Docker. Untuk itu, dibutuhkan arsitektur yang mampu mengintegrasikan manajemen autentikasi pengguna, alokasi kontainer secara dinamis, serta orkestrasi workload komputasi berbasis GPU maupun CPU.

Langkah awal dalam perancangan adalah merancang sistem *service discovery* yang berfungsi sebagai pusat pengumpulan dan penyimpanan status sumber daya dari seluruh node yang tersedia dalam *cluster*. Data ini mencakup informasi penggunaan CPU, RAM, informasi apakah node memiliki GPU atau tidak, serta jumlah *container* aktif, yang dikirim secara periodik oleh agent service dari masing-masing node. Informasi ini disimpan ke dalam basis data in-memory Redis yang akan digunakan untuk mendukung pengambilan keputusan secara real-time saat pemilihan node dilakukan.

Selanjutnya, dilakukan integrasi antara JupyterHub sebagai antarmuka utama pengguna dan DockerSpawner yang telah dimodifikasi untuk mendukung pendistribusian JupyterLab *multi-node*. Dengan adanya integrasi ini, setiap permintaan pengguna akan diproses melalui skema load balancing, yang kemudian menentukan node terbaik untuk menjalankan *environment* JupyterLab. Skor dihitung berdasarkan tingkat utilisasi CPU, RAM, dan jumlah *container* aktif.

Gambar 3.1 di bawah ini menggambarkan secara keseluruhan hubungan antar komponen sistem. Diagram tersebut memperlihatkan arsitektur sistem yang dirancang, mulai dari proses pelaporan status node oleh Agent Service, pemrosesan dan pengambilan keputusan di Discovery Service, hingga peluncuran kontainer pengguna di JupyterHub yang diintegrasikan dengan Jupyter Enterprise Gateway (JEG).



Gambar 3.1: Arsitektur Penelitian

Adapun komponen utama yang membentuk arsitektur sistem ini terdiri atas:

### 3.1.1 Service Discovery

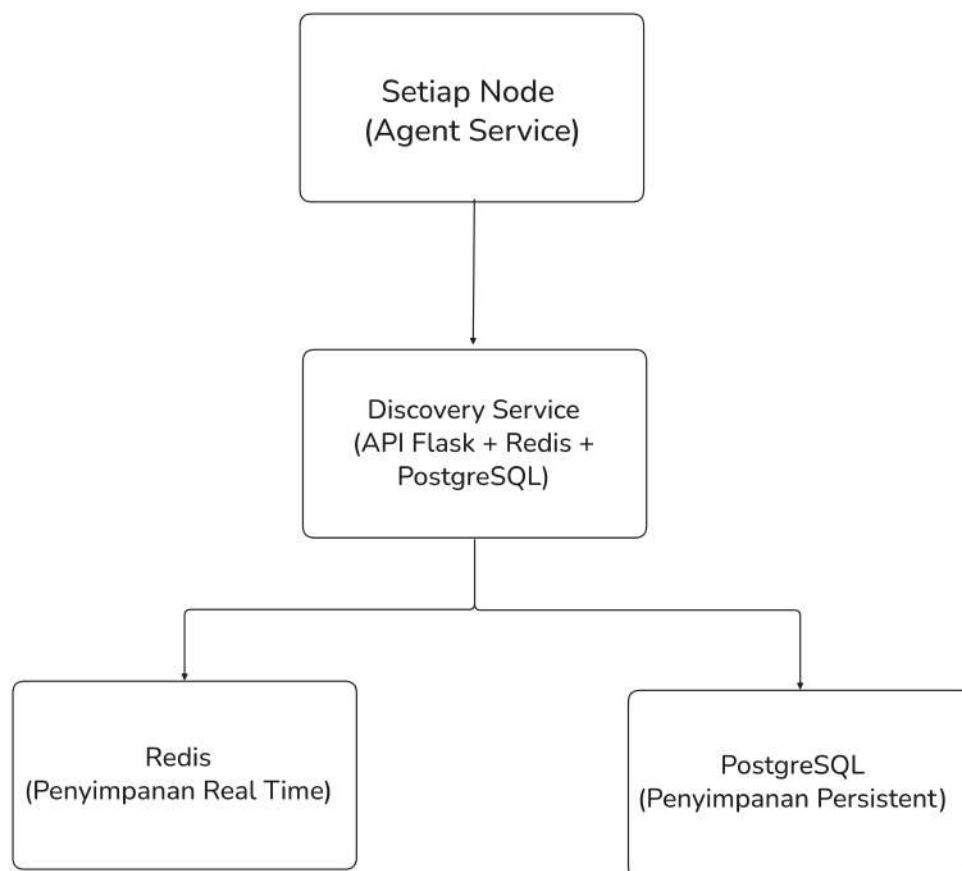
Service Discovery merupakan komponen tambahan yang berfungsi sebagai pusat informasi dan pengendali dalam sistem distribusi beban kerja kernel. Komponen ini terdiri dari dua bagian utama, yaitu Discovery API, yang berjalan sebagai service pusat, dan agent yang berjalan secara

terpisah di setiap node worker. Peran utamanya adalah memastikan bahwa pemilihan node tempat eksekusi kernel dilakukan secara dinamis, adaptif, dan berbasis data aktual dari masing-masing node.

Setiap agent di node worker bertugas untuk memonitor kondisi sistem seperti penggunaan CPU, RAM, jumlah container aktif (baik JupyterLab maupun Ray), serta ketersediaan GPU. Informasi ini dikirimkan secara periodik ke Discovery API menggunakan HTTP POST, dan disimpan dalam basis data Redis dengan mekanisme TTL (time-to-live) agar status node tetap terkini. Data ini kemudian digunakan oleh load balancer internal di Discovery API untuk menilai skor beban dari masing-masing node menggunakan algoritma seperti Weighted Round Robin dan Least Connection.

Discovery API menyediakan endpoint `/available-nodes` dan `/jupyterhub-nodes` yang dikonsumsi langsung oleh spawner JupyterHub saat proses spawn dijalankan. Pemilihan node dilakukan secara otomatis berdasarkan skor beban yang diperoleh dari data real-time. Selain itu, sistem ini juga memungkinkan filter berdasarkan fitur node (misalnya ketersediaan GPU) sesuai kebutuhan profil pengguna. Arsitektur ini memungkinkan distribusi kernel secara merata dan efisien, serta membantu mencegah overload pada node tertentu.

Keberadaan Service Discovery ini menjadi kunci utama dalam arsitektur multi-node yang diusulkan, karena menjembatani kebutuhan akan load balancing berbasis sumber daya dengan sistem orkestrasi yang fleksibel dan terintegrasi penuh dengan JupyterHub dan Enterprise Gateway.



Gambar 3.2: Diagram alir service discovery





Gambar 3.3: Diagram alir service discovery dalam menampilkan ketersediaan *node*

Gambar 3.3 merupakan proses *node* (Agent Service) dalam melaporkan informasi ke Service Discovery melalui *endpoint* **/register-node**, kemudian informasi yang dilaporkan akan disimpan ke dalam *database* dan ditampilkan melalui *endpoint* **/available-nodes**. Nantinya data yang ditampilkan di **/available-nodes** akan diintegrasikan dengan JupyterHub untuk memilih *node* yang terbaik untuk menjalankan *container* JupyterLab.

### 3.1.2 Service Agent

Service Agent merupakan komponen yang berjalan secara periodik di setiap *node* dalam kluster. Tugas utama Agent adalah memantau kondisi sistem secara lokal, kemudian mengirimkan informasi tersebut ke Discovery Service melalui *endpoint* **/register-node**. Informasi yang dikirim mencakup:

- Informasi jumlah CPU, kapasitas memori dan penyimpanan, serta tingkat penggunaan (usage) masing-masing sumber daya secara real-time.
- Deteksi keberadaan GPU (terutama NVIDIA) beserta detail spesifikasinya seperti kapasitas memori dan tingkat utilisasi.
- Informasi tambahan seperti *hostname*, alamat IP, dan metadata *node* lainnya.

Agent dirancang dalam bentuk *container* mandiri berbasis Python yang berjalan otomatis sejak *node* aktif. Dengan mengandalkan pustaka seperti *psutil*, *gpustat*, dan Docker SDK, Agent mampu menangkap informasi sistem secara akurat. Interval pengiriman data diatur setiap 15 detik agar status *node* tetap mutakhir di Redis tanpa membebani sumber daya secara signifikan.

Komponen ini sangat krusial dalam menjaga keakuratan data load balancing, karena JupyterHub akan memilih *node* berdasarkan data yang dikumpulkan oleh Agent. Dengan adanya Agent, sistem dapat secara otomatis mengetahui jika suatu *node* mengalami kelebihan beban, tidak tersedia, atau sedang dalam kondisi idle.

### 3.1.3 JupyterHub

JupyterHub bertindak sebagai sistem pusat autentikasi dan pengelola sesi pengguna dalam arsitektur sistem ini. Platform ini memungkinkan setiap pengguna untuk memulai sesi JupyterLab pribadi yang berjalan di dalam *container* terisolasi, memastikan keamanan dan independensi lingkungan kerja antar pengguna. Untuk menangani arsitektur *multi-node*, digunakan *spawner* kustom bernama **MultiNodeSpawner**, yang merupakan hasil modifikasi dari **DockerSpawner**. *Spawner* ini terintegrasi dengan Discovery API dan bertugas untuk memilih *node*

terbaik secara otomatis berdasarkan skor beban seperti penggunaan CPU, RAM, jumlah *container* aktif, serta ketersediaan GPU. Setelah node dipilih, spawner akan mengatur parameter *container* secara otomatis, termasuk pemetaan IP dan *port* yang sesuai, pemilihan *image* Docker sesuai profil yang dipilih pengguna, serta konfigurasi URL *kernel* agar terhubung dengan Enterprise Gateway. Seluruh proses ini berjalan secara transparan bagi pengguna, yang cukup memilih profil komputasi dan *node* melalui antarmuka yang telah dikustomisasi. Hasilnya adalah sistem yang tidak hanya efisien secara teknis, namun juga mudah diakses oleh pengguna dengan latar belakang teknis yang beragam.

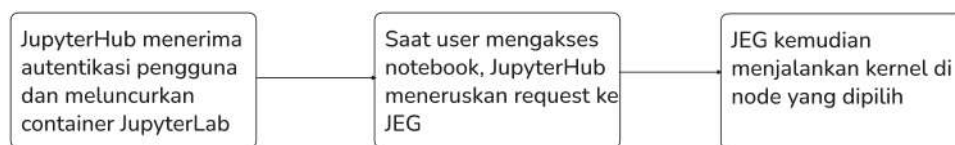
### 3.1.4 Jupyter Enterprise Gateway

Jupyter Enterprise Gateway (JEG) merupakan komponen tambahan yang dirancang untuk meningkatkan skalabilitas dan fleksibilitas dalam eksekusi kernel Jupyter di lingkungan terdistribusi. Dalam penelitian ini, JEG digunakan untuk memisahkan proses eksekusi kernel dari JupyterHub dan JupyterLab itu sendiri, sehingga memungkinkan kernel dijalankan pada node-node yang terpisah berdasarkan ketersediaan sumber daya.

Dalam arsitektur sistem yang diusulkan, JupyterHub tidak lagi secara langsung meluncurkan kernel ke dalam container JupyterLab. Sebaliknya, ketika pengguna membuka notebook, permintaan kernel dikirim ke Jupyter Enterprise Gateway, yang bertugas memilih node tempat kernel akan dieksekusi. Gateway ini kemudian menginisialisasi kernel di node target melalui protokol ZeroMQ (ZMQ) untuk komunikasi backend, dan menjaga koneksi dengan pengguna menggunakan WebSocket Secure (WSS) untuk saluran interaktif.

Dengan memanfaatkan komunikasi asinkron berbasis ZMQ, proses antara JEG dan kernel menjadi efisien dan ringan, sementara protokol WSS menjaga konektivitas real-time antara JupyterLab dan kernel terdistribusi. Pemisahan ini memberikan fleksibilitas tinggi dalam pengelolaan beban kerja, karena node-node worker dapat ditambahkan atau dikurangi secara dinamis sesuai kebutuhan. Selain itu, arsitektur ini juga mendukung skalabilitas horizontal, sehingga cocok diterapkan dalam sistem dengan jumlah pengguna atau kebutuhan komputasi yang tinggi.

Implementasi JEG dalam sistem ini juga mendukung pengaturan sumber daya berbasis container, sehingga setiap kernel dapat dijalankan dalam lingkungan yang dikonfigurasi khusus, termasuk akses ke GPU atau batasan memori tertentu. Kombinasi antara JupyterHub, JEG, dan Discovery API menghasilkan sistem orkestrasi kernel yang adaptif, efisien, dan mudah dikembangkan di lingkungan terdistribusi.



Gambar 3.4: Alur komunikasi JEG dan JupyterHub

## 3.2 Implementasi Sistem

Implementasi sistem terdiri dari beberapa komponen utama yang saling terintegrasi, sebagai berikut:

### 3.2.1 Implementasi Service Discovery

Discovery Service merupakan komponen pusat dalam sistem yang bertugas menerima, menyimpan, dan menyediakan informasi status sumber daya dari setiap node GPU. Layanan ini dibangun dengan framework Flask REST API dan mengimplementasikan pendekatan penyimpanan hybrid menggunakan Redis dan PostgreSQL. Redis digunakan untuk data real-time dengan TTL (time-to-live), sedangkan PostgreSQL menyimpan data historis dan metadata yang lebih persisten.

Pada bagian ini akan dijelaskan struktur proyek, konfigurasi sistem, serta implementasi fitur-fitur utama dari layanan Discovery Service secara teknis.

#### 3.2.1.1 Arsitektur Aplikasi

Struktur proyek Discovery Service disusun secara modular dengan pembagian tanggung jawab yang jelas. Tabel 3.1 menjelaskan file dan direktori utama:

Tabel 3.1: Struktur Direktori Discovery Service

Nama File/Folder	Deskripsi
app.py	Titik masuk aplikasi Flask.
config.py	Konfigurasi environment dan database.
redis_client.py	Utilitas koneksi Redis.
Dockerfile	Definisi image Docker.
docker-compose.yml	Orkestrasi Redis, PostgreSQL, dan Flask.
init.sql	Skrip inisialisasi database PostgreSQL.
redis.conf	Konfigurasi Redis.
models/	ORM SQLAlchemy untuk Node, Profile, dsb.
routes/	Endpoint API untuk node dan profile.
services/	Logika bisnis seperti pendaftaran node.
utils/	Load balancer dan skoring node.

#### 3.2.1.2 Inisialisasi Proyek dan Registrasi Layanan

Selain menginisialisasi konfigurasi dasar seperti CORS, database, dan blueprint, file **app.py** juga mencakup integrasi awal dengan basis data PostgreSQL dan Redis. Salah satu endpoint bawaan adalah **/health-check** yang digunakan untuk memastikan apakah API telah berjalan dan memverifikasi status koneksi ke kedua database tersebut secara real-time. Redis digunakan melalui kelas **RedisService** untuk pengecekan konektivitas dan pengelolaan data status node yang *volatile*.

Selain itu, fungsi **run\_periodic\_task** digunakan untuk menjalankan tugas latar belakang

yang membersihkan node-node yang tidak aktif berdasarkan data dari Redis. Hal ini meningkatkan reliabilitas data yang tersimpan dan mengurangi beban layanan.

---

```
1 FUNCTION create_app():
2     INITIALIZE Flask app
3     LOAD configuration from Config
4     INITIALIZE extensions:
5         - CORS
6         - SQLAlchemy database
7         - Flask Migrate
8
9     REGISTER blueprints:
10        - node_bp
11        - profile_bp
12
13    DEFINE route "/health-check":
14        RETURN JSON status (ok) with database connection status
15
16    WITH app context:
17        CREATE all database tables
18        INITIALIZE default profiles
19
20    RETURN app
21 END FUNCTION
22
23 FUNCTION run_periodic_tasks(app):
24    START background thread:
25        WHILE True:
26            CALL mark_nodes_inactive() to cleanup inactive nodes
27            SLEEP 300 seconds
28 END FUNCTION
29
30 IF __name__ == '__main__':
31     app = create_app()
32     CALL run_periodic_tasks(app)
33     RUN app on host 0.0.0.0 port 15002 (debug mode)
34 END IF
```

---

Kode Sumber 3.1: Kode Semu untuk app.py

### 3.2.1.3 Integrasi dengan Basis Data

Discovery Service menggunakan dua jenis sistem basis data untuk mendukung performa dan keandalan layanan: **PostgreSQL** sebagai basis data relasional permanen dan **Redis** sebagai penyimpanan data sementara (in-memory) untuk status sistem.

PostgreSQL diakses melalui ORM SQLAlchemy yang telah terhubung di dalam file app.py menggunakan db.init\_app(app). Tabel-tabel utama yang dimodelkan dalam sistem ini antara lain:

- **Node:** Menyimpan informasi node seperti hostname, kapasitas CPU, RAM, dan keberadaan GPU.
- **NodeMetric:** Menyimpan riwayat pemantauan beban node seperti penggunaan CPU, memori, jumlah kontainer aktif, dan skoring beban.

- **Profile:** Mendefinisikan konfigurasi profil pengguna yang menentukan kebutuhan resource.
- **NodeSelection:** Mencatat hasil seleksi node berdasarkan profil dan pengguna.

Semua model didefinisikan secara modular dalam direktori `models/`. Skema database dapat diinisialisasi dan dimigrasi menggunakan *library* `Flask-Migrate`.

---

```

1 CLASS Node:
2     ATTRIBUTE id : Integer
3     ATTRIBUTE hostname : String(255)
4     ATTRIBUTE ip : String(45)
5     ATTRIBUTE cpu_cores : Integer
6     ATTRIBUTE ram_gb : Float
7     ATTRIBUTE has_gpu : Boolean = False
8     ATTRIBUTE gpu_info : JSON
9     ATTRIBUTE is_active : Boolean
10    ATTRIBUTE max_containers : Integer
11    ATTRIBUTE created_at : Datetime
12    ATTRIBUTE updated_at : Datetime
13 END CLASS

```

---

Kode Sumber 3.2: Kode Semu untuk Model Node

Tabel 3.2: Kolom dan Tipe Data Model Node

Nama Kolom	Tipe Data
id	INTEGER (PRIMARY KEY)
hostname	STRING(255)
ip	STRING(45)
cpu_cores	INTEGER
ram_gb	FLOAT
has_gpu	BOOLEAN
gpu_info	JSON
is_active	BOOLEAN
max_containers	INTEGER
created_at	DATETIME
updated_at	DATETIME

---

```

1 CLASS NodeSelection:
2     ATTRIBUTE id : Integer
3     ATTRIBUTE profile_id : Integer
4     ATTRIBUTE user_id : String(255)
5     ATTRIBUTE session_id : String(255)
6     ATTRIBUTE selected_nodes : JSON
7     ATTRIBUTE selection_reason : String(50)
8     ATTRIBUTE created_at : Datetime
9 END CLASS

```

---

Kode Sumber 3.3: Kode Semu untuk Model NodeSelection

Tabel 3.3: Kolom dan Tipe Data Model NodeSelection

Nama Kolom	Tipe Data
id	INTEGER (PRIMARY KEY)
profile_id	INTEGER
user_id	STRING(255)
session_id	STRING(255)
selected_nodes	JSON
selection_reason	STRING(50)
created_at	DATETIME

---

```
1 CLASS NodeMetric:
2     ATTRIBUTE id : Integer
3     ATTRIBUTE node_id : Integer
4     ATTRIBUTE cpu_usage_percent : Float
5     ATTRIBUTE memory_usage_percent : Float
6     ATTRIBUTE disk_usage_percent : Float
7     ATTRIBUTE active_jupyterlab : Integer
8     ATTRIBUTE active_ray : Integer
9     ATTRIBUTE total_containers : Integer
10    ATTRIBUTE load_score : Float
11    ATTRIBUTE recorded_at : Datetime
12 END CLASS
```

---

Kode Sumber 3.4: Kode Semu untuk Model NodeMetric

Tabel 3.4: Kolom dan Tipe Data Model NodeMetric

Nama Kolom	Tipe Data
id	INTEGER (PRIMARY KEY)
node_id	INTEGER
cpu_usage_percent	FLOAT
memory_usage_percent	FLOAT
disk_usage_percent	FLOAT
active_jupyterlab	INTEGER
active_ray	INTEGER
total_containers	INTEGER
load_score	FLOAT
recorded_at	DATETIME

Redis digunakan untuk menyimpan status terkini node yang dilaporkan oleh agent secara periodik. Redis ini tidak menyimpan data permanen, tetapi digunakan untuk menyimpan metrik real-time seperti CPU, RAM, dan disk usage

Koneksi ke Redis dilakukan melalui file `redis_client.py` menggunakan *connection pool* untuk efisiensi koneksi. File ini diakses melalui kelas `RedisService` pada `fileredis_service.py`.

---

```
1 SET REDIS_HOST = getenv("REDIS_HOST", "localhost")
2 SET REDIS_PORT = getenv("REDIS_PORT", 6379) AS Integer
3 SET REDIS_PASSWORD = getenv("REDIS_PASSWORD")
4 SET REDIS_EXPIRE_SECONDS = getenv("REDIS_EXPIRE_SECONDS", 45) AS Integer
5
6 INITIALIZE ConnectionPool with:
7     host = REDIS_HOST
8     port = REDIS_PORT
9     password = REDIS_PASSWORD
10    decode_responses = True
11
12 INITIALIZE RedisClient with ConnectionPool
```

---

Kode Sumber 3.5: Kode Semu untuk Koneksi Redis Menggunakan ConnectionPool

Agent akan mengirimkan data dalam interval tertentu, dan informasi tersebut disimpan sementara dalam Redis menggunakan TTL selama 45 detik. Berikut contoh potongan penyimpanan data pada saat registrasi node:

---

```
1 CALL RedisClient.set(
2     key = "node:{hostname}:info",
3     value = JSON.stringify(data),
4     expire = Config.REDIS_EXPIRE_SECONDS
5 )
```

---

Kode Sumber 3.6: Kode Semu Penyimpanan Data Node ke Redis dengan TTL

### 3.2.1.4 Seleksi Node

Discovery Service menggunakan pendekatan modular dalam proses seleksi node, yang diimplementasikan dalam file `load_balancer.py` pada direktori *utils/*. Pemilihan node dilakukan berdasarkan algoritma yang dapat disesuaikan, seperti *round robin*, *best fit*, dan *random selection*, dengan *round robin* sebagai metode default untuk mendistribusikan beban kerja antar node secara merata.

Sebelum pemilihan dilakukan, setiap node dihitung nilai beban-nya melalui fungsi di bawah ini, yang berada pada file `scoring.py`. Fungsi ini menghitung skor berdasarkan kombinasi tingkat utilisasi CPU dan memori. Node yang melebihi ambang batas penggunaan sumber daya akan dikenakan penalti tambahan, sehingga menghasilkan skor yang lebih tinggi dan cenderung tidak diprioritaskan.

---

```
1 FUNCTION calculate_node_score(node_data) RETURNS Float:
2     SET cpu_usage = node_data["cpu_usage_percent"] OR 100
3     SET memory_usage = node_data["memory_usage_percent"] OR 100
4
5     SET score = (cpu_usage * CPU_WEIGHT) + (memory_usage * MEMORY_WEIGHT)
6
7     IF cpu_usage > 90 OR memory_usage > 90 THEN
8         score = score + HEAVY_PENALTY
```

---

```

9     ELSE IF cpu_usage > 80 OR memory_usage > 80 THEN
10         score = score + MEDIUM_PENALTY
11     END IF
12
13     RETURN Round(score, 2)
14 END FUNCTION

```

---

Kode Sumber 3.7: Kode Semu Fungsi Perhitungan Skor Node

---

Selain itu, fungsi `select_nodes_by_algorithm()` digunakan untuk memilih node terbaik sesuai algoritma yang ditentukan, sedangkan `distribute_load()` digunakan untuk mendistribusikan workload berdasarkan kapasitas maksimal per node.

### 3.2.1.5 Konfigurasi Environment

Discovery Service menggunakan pendekatan berbasis konfigurasi eksternal agar sistem dapat dengan mudah dijalankan di berbagai lingkungan seperti *development*, maupun *production*. Semua pengaturan disatukan dalam satu *file* `config.py` yang memanfaatkan *library* `python-dotenv` untuk membaca variabel dari file `.env`.

---

```

1 CLASS Config:
2     ATTRIBUTE SECRET_KEY = getenv("SECRET_KEY", "secret-service-1111")
3     ATTRIBUTE DEBUG = getenv("DEBUG", "True") == "true"
4
5     // Database Configuration
6     ATTRIBUTE POSTGRES_HOST = getenv("POSTGRES_HOST", "localhost")
7     ATTRIBUTE POSTGRES_PORT = getenv("POSTGRES_PORT", "5432")
8     ATTRIBUTE POSTGRES_DB = getenv("POSTGRES_DB", "discovery")
9     ATTRIBUTE POSTGRES_USER = getenv("POSTGRES_USER", "postgres")
10    ATTRIBUTE POSTGRES_PASSWORD = getenv("POSTGRES_PASSWORD", "postgres")
11
12    ATTRIBUTE SQLALCHEMY_DATABASE_URI =
13        "postgresql://" + POSTGRES_USER + ":" + POSTGRES_PASSWORD +
14        "@" + POSTGRES_HOST + ":" + POSTGRES_PORT + "/" + POSTGRES_DB
15
16    ATTRIBUTE SQLALCHEMY_TRACK_MODIFICATIONS = False
17    ATTRIBUTE SQLALCHEMY_ECHO = getenv("SQLALCHEMY_ECHO", "false") == "↵
18        true"
19
20    // Redis Configuration
21    ATTRIBUTE REDIS_HOST = getenv("REDIS_HOST", "localhost")
22    ATTRIBUTE REDIS_PORT = getenv("REDIS_PORT", 6379) AS Integer
23    ATTRIBUTE REDIS_PASSWORD = getenv("REDIS_PASSWORD", "redis@pass")
24    ATTRIBUTE REDIS_EXPIRE_SECONDS = getenv("REDIS_EXPIRE_SECONDS", 45) ↵
25        AS Integer
26
27    // Load Balancer Thresholds
28    ATTRIBUTE DEFAULT_MAX_CPU_USAGE = 80.0
29    ATTRIBUTE DEFAULT_MAX_MEMORY_USAGE = 85.0
30    ATTRIBUTE STRICT_MAX_CPU_USAGE = 60.0
31    ATTRIBUTE STRICT_MAX_MEMORY_USAGE = 60.0
32    ATTRIBUTE STRICT_MAX_CONTAINERS = 5
33
34    // Scoring Weights & Penalties
35    ATTRIBUTE CPU_WEIGHT = 0.8
36    ATTRIBUTE MEMORY_WEIGHT = 0.8
37    ATTRIBUTE HEAVY_PENALTY = 80

```



```

36     ATTRIBUTE MEDIUM_PENALTY = 20
37 END CLASS

```

Kode Sumber 3.8: Kode Semu untuk `config.py`

Seluruh konfigurasi di atas bersifat dinamis dan dapat disesuaikan melalui file `.env` tanpa perlu mengubah kode Python. Contoh isi file konfigurasi lingkungan dapat dilihat pada Tabel 3.5 berikut:

Tabel 3.5: Contoh Isi File `.env` dari *Discovery Service*

Variabel	Deskripsi
FLASK_DEBUG=True	Mengaktifkan mode debug pada aplikasi Flask.
SECRET_KEY=secret-service111111	Kunci rahasia untuk keperluan autentikasi Flask.
POSTGRES_HOST=127.0.0.1	Alamat host untuk koneksi ke database PostgreSQL.
POSTGRES_PORT=5432	Port yang digunakan PostgreSQL.
POSTGRES_DB=voyager	Nama database utama yang digunakan.
POSTGRES_USER=postgres	Nama pengguna untuk mengakses PostgreSQL.
POSTGRES_PASSWORD=postgres	Password pengguna PostgreSQL.
REDIS_HOST=127.0.0.1	Alamat host untuk server Redis.
REDIS_PORT=6379	Port Redis yang digunakan.
REDIS_PASSWORD=redis@pass	Password autentikasi ke Redis.
REDIS_EXPIRE_SECONDS=45	Waktu kedaluwarsa (dalam detik) untuk data Redis.

Dengan struktur seperti ini, sistem dapat dengan mudah dipindahkan antar server atau dijalankan dalam konteks kontainer Docker tanpa harus mengubah kode utama aplikasi.

### 3.2.1.6 Deployment Service Discovery dengan Docker

Untuk memudahkan proses deployment dan reproduksibilitas lingkungan, *Discovery Service* dikemas dalam sebuah image menggunakan Docker. Layanan ini selanjutnya diatur dengan Docker Compose untuk menjalankan seluruh komponen (Flask API, Redis, PostgreSQL) secara terorkestrasi.

Dockerfile berikut akan digunakan untuk membangun *container service discovery*, menginstal dependensi dari `requirements.txt`, dan menjalankan `app.py` sebagai aplikasi utama.

```

1 PROCEDURE BuildDockerImage
2
3     // Inisialisasi Konfigurasi Dasar
4     SET BaseImage TO "python:3.12-slim"
5     SET WorkingDirectory TO "/app"
6
7     // Konfigurasi Environment Variables
8     SET EnvVariable "PYTHONDONTWRITEBYTECODE" TO "1"
9     SET EnvVariable "PYTHONUNBUFFERED" TO "1"

```

```

10     SET EnvVariable "TZ" TO "Asia/Jakarta"
11
12     // Proses Instalasi Dependensi
13     COPY "requirements.txt" FROM source TO "/app/"
14     EXECUTE "pip install --no-cache-dir -r requirements.txt"
15
16     // Salin Kode Aplikasi ke WorkingDirectory
17     COPY all_files FROM source TO "/app/"
18
19     // Konfigurasi Jaringan dan Eksekusi
20     EXPOSE Port 15002
21     SET DefaultCommand TO ["python", "app.py"]
22
23 END PROCEDURE

```

---

### Kode Sumber 3.9: Kode Semu Proses Pembangunan Image Docker

Untuk menjalankan layanan ini secara bersamaan dengan Redis dan PostgreSQL, digunakan `docker-compose.yml` berikut:

---

```

1 COMPOSE VERSION: 3.8
2
3 DEFINE SERVICES:
4
5     SERVICE discovery:
6         BUILD from Dockerfile in current context
7         CONTAINER NAME = discovery-api
8         RESTART POLICY = unless-stopped
9         EXPOSE PORTS: 15002:15002
10
11     SET ENVIRONMENT VARIABLES:
12         POSTGRES_HOST = postgres
13         POSTGRES_PORT = 5432
14         POSTGRES_DB = voyager
15         POSTGRES_USER = postgres
16         POSTGRES_PASSWORD = postgres
17
18         REDIS_HOST = redis
19         REDIS_PORT = 16379
20         REDIS_PASSWORD = redis@pass
21         REDIS_EXPIRE_SECONDS = 45
22
23         API_HOST = 0.0.0.0
24         API_PORT = 15002
25         DEBUG = True
26         SECRET_KEY = secret-service111111
27
28     DEPENDS ON: postgres, redis
29     NETWORKS: discovery-network
30
31     SERVICE postgres:
32         IMAGE = postgres:14-alpine
33         CONTAINER NAME = postgres
34         RESTART POLICY = unless-stopped
35         EXPOSE PORTS: 5432:5432
36
37     SET ENVIRONMENT VARIABLES:

```

```

38     POSTGRES_DB = ds
39     POSTGRES_USER = postgres
40     POSTGRES_PASSWORD = postgres
41     POSTGRES_INITDB_ARGS = --encoding=UTF-8
42     TZ = Asia/Jakarta
43
44     VOLUMES:
45         postgres_data -> /var/lib/postgresql/data
46         ./init.sql -> /docker-entrypoint-initdb.d/init.sql
47
48     HEALTHCHECK:
49         COMMAND = pg_isready -U postgres -d discovery_db
50         INTERVAL = 10s
51         TIMEOUT = 5s
52         RETRIES = 5
53         START_PERIOD = 30s
54
55     NETWORKS: discovery-network
56
57     SERVICE redis:
58         IMAGE = redis:7-alpine
59         CONTAINER NAME = redis
60         RESTART POLICY = unless-stopped
61         EXPOSE PORTS: 16379:16379
62
63     VOLUMES:
64         redis_data -> /data
65         ./redis.conf -> /usr/local/etc/redis/redis.conf
66
67     COMMAND: redis-server /usr/local/etc/redis/redis.conf
68
69     SET ENVIRONMENT VARIABLES:
70         TZ = Asia/Jakarta
71
72     HEALTHCHECK:
73         COMMAND = redis-cli -p 16379 ping
74         INTERVAL = 10s
75         TIMEOUT = 3s
76         RETRIES = 3
77
78     NETWORKS: discovery-network
79
80     DEFINE VOLUMES:
81         postgres_data
82         redis_data
83
84     DEFINE NETWORKS:
85         discovery-network (driver: bridge)

```

---

Kode Sumber 3.10: Kode Semu untuk docker-compose.yml

docker-compose.yml mendefinisikan tiga layanan utama: discovery, postgres, dan redis, yang saling terhubung melalui jaringan internal discovery-network. Untuk menjalankan seluruh services, gunakan perintah:

---

```
1 docker-compose up -d --build
```

---

Kode Sumber 3.11: Menjalankan Discovery Service via Docker Compose

### 3.2.2 Implementasi Service Agent

Setelah layanan Discovery Service diimplementasikan, sistem memerlukan komponen tambahan yang berjalan secara periodik di setiap node. Komponen ini disebut sebagai *Agent Service*. Agent bertanggung jawab untuk mengumpulkan informasi sistem dan mengirimkannya secara berkala ke *endpoint /register-node* pada *Discovery API*. Informasi tersebut mencakup pemanfaatan CPU, memori, *disk*, deteksi GPU, serta jumlah *container* yang sedang aktif. Bagian ini akan menjelaskan konfigurasi dari Agent Service dan *deployment* menggunakan Docker secara lebih mendalam.

#### 3.2.2.1 Implementasi Pengumpulan Data

Agent dikembangkan menggunakan kode Python yang berjalan sebagai *container* pada setiap *node*. Agent ini dirancang agar dapat mengirimkan informasi *node* yang diperlukan setiap 15 detik.

Agent diimplementasikan sebagai Docker *container* terpisah di setiap node. Agent secara berkala mengumpulkan informasi sistem dan mengirimkannya ke Discovery API melalui *endpoint /register-node*. Seluruh proses berlangsung setiap 15 detik, memastikan bahwa data yang dikirim tetap *up-to-date*.

Fungsi utama agent dimulai dari **register()**, seperti ditunjukkan pada Kode Sumber 3.12. Fungsi ini bertugas mengumpulkan data menggunakan **collect\_node\_info()** dan mengirimkannya ke API.

---

```
1 FUNCTION register():
2     SET payload = collect_node_info()
3
4     IF payload IS NOT EMPTY THEN
5         CALL HTTP POST to DISCOVERY_URL with JSON payload
6     END IF
7 END FUNCTION
```

---

Kode Sumber 3.12: Kode Semu Fungsi Register Agent

Fungsi **collect\_node\_info()** bertanggung jawab untuk membaca informasi hardware dan beban kerja node. Data yang dikumpulkan meliputi:

- Penggunaan CPU, memori, dan disk saat ini.
- Informasi jumlah *container* (terutama JupyterLab).
- Deteksi GPU NVIDIA.

---

```
1 FUNCTION collect_node_info():
2
3     SET hostname = GET system hostname
4     SET ip_address = GET primary IP address of system
5
6     SET ram_gb = total RAM in GB (rounded to 2 decimals)
7     SET cpu_usage = current CPU usage percent (measured over 1 second)
8     SET memory = current memory usage summary
9     SET disk = disk usage summary for root directory "/"
10
11 END FUNCTION
```

---

---

### Kode Sumber 3.13: Kode Semu Kumpulan Informasi Sistem oleh Agent

Agent juga menghitung jumlah container Jupyter yang berjalan dengan membaca nama dan image-nya. Hal ini dilakukan oleh fungsi **get\_container\_info()**, yang akan mengenali JupyterLab.

---

```
1 FUNCTION get_container_info()
2   INITIALIZE jupyterlab_count = 0
3
4   SET container_list TO get_all_running_containers()
5
6   FOR EACH container IN container_list
7     IF "jupyter" is in container.name OR "jupyter" is in container.image_tags THEN
8       INCREMENT jupyterlab_count
9     END IF
10  END FOR
11
12  RETURN jupyterlab_count
13 END FUNCTION
```

---

### Kode Sumber 3.14: Deteksi Container JupyterLab

Untuk mendeteksi keberadaan GPU, agent menggunakan pustaka atau *library* **gpustat**. Jika GPU NVIDIA tersedia, maka informasi seperti penggunaan memori, utilisasi, dan *load* GPU akan dikirimkan.

---

```
1 FUNCTION get_gpu_stats()
2   INITIALIZE gpu_info_list as an empty list
3
4   TRY
5     SET stats TO query_gpu_information()
6
7     FOR EACH gpu IN stats.gpus
8       APPEND {
9         "name": gpu.name,
10        "memory_used_mb": gpu.memory_used,
11        "utilization_gpu_percent": gpu.utilization
12      } TO gpu_info_list
13   END FOR
14
15   CATCH No GPU Found Error
16   END TRY
17
18   RETURN gpu_info_list
19 END FUNCTION
```

---

### Kode Sumber 3.15: Deteksi GPU Menggunakan gpustat

Akhirnya, agent akan menjalankan proses ini dalam *loop* tak hingga, mengirimkan data ke API setiap 15 detik. Hal ini memungkinkan Discovery Service selalu memiliki data terbaru untuk pengambilan keputusan.

---

```
1 IF this script is executed as main program THEN
2   WHILE True DO
```

```

3         CALL register()
4         WAIT 15 seconds
5     END WHILE
6 END IF

```

---

Kode Sumber 3.16: Kode Semu Loop Registrasi Agent Tiap 15 Detik

### 3.2.2.2 Konfigurasi Remote Docker Daemon

Agar proses peluncuran container JupyterLab dari JupyterHub dapat dilakukan secara *remote* ke *node* yang terpilih, maka Docker *daemon* pada setiap *node* perlu dikonfigurasi agar dapat menerima koneksi jaringan melalui protokol TCP. Hal ini memungkinkan komponen Spawner pada JupyterHub untuk menjalankan perintah **docker run** terhadap *daemon* Docker di *node* tujuan, yang sebelumnya telah dipilih melalui Discovery Service.

Secara default, Docker hanya melayani koneksi melalui UNIX socket lokal (/var/run/docker.sock). Untuk mengaktifkan akses remote, pastikan telah menginstal Docker sebelumnya. Untuk konfigurasi dilakukan dengan langkah-langkah sebagai berikut:

---

```

1 # Buka file konfigurasi Docker
2 sudo nano /lib/systemd/system/docker.service
3
4 # Ubah baris "ExecStart" menjadi:
5 ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375
6
7 # Reload dan restart service Docker
8 sudo systemctl daemon-reload
9 sudo systemctl restart docker.service

```

---

Kode Sumber 3.17: Langkah Konfigurasi Docker untuk Akses Remote

Setelah konfigurasi ini selesai, JupyterHub dapat mengakses Docker *daemon* pada *node* dengan alamat `tcp://<ip-node>:2375`, dan menjalankan *container* JupyterLab UI di *node* tersebut secara langsung. Ini merupakan bagian penting dari sistem orkestrasi *multi-node* yang dibangun dalam penelitian ini.

### 3.2.2.3 Deployment Agent

Agar dapat berjalan secara independen di setiap *node*, Agent dibungkus ke dalam sebuah *container* menggunakan Docker. Hal ini memungkinkan *deployment* yang konsisten di seluruh lingkungan tanpa bergantung pada konfigurasi sistem *host*. Kode sumber 3.18 menunjukkan isi file Dockerfile yang digunakan untuk membangun *image* Agent.

*Base image* yang digunakan pada Dockerfile adalah `python:3.12-slim` untuk memastikan image tetap ringan. *Working Directory* di-set ke `/app`, dan seluruh kode serta dependensi diinstal melalui `file requirements.txt`. *Command* akhir akan menjalankan file `agent.py`.

---

```

1 BUILD IMAGE:
2     BASE IMAGE = python:3.12-slim
3
4     SET ENVIRONMENT VARIABLES:
5         PYTHONDONTWRITEBYTECODE = 1
6         PYTHONUNBUFFERED = 1
7         TZ = Asia/Jakarta
8
9     SET WORKING DIRECTORY = /app

```

---

```

10
11     COPY file requirements.txt to /app
12
13     RUN pip install requirements from requirements.txt (no cache)
14
15     COPY all files from build context to /app
16
17     EXPOSE PORT 15002
18
19     SET DEFAULT COMMAND to run:
20         "python agent.py"
21 END BUILD

```

---

#### Kode Sumber 3.18: Dockerfile untuk Membangun Agent Service

Selanjutnya *agent* akan di-*build* menjadi Docker image dengan nama *danielcrsth0/agent:1.1* dan akan di-*push* atau dikirim ke Docker *registry*, tujuan dari *image agent* di-*push* ke *registry* adalah memudahkan penyebaran *image* di setiap *node*:

---

```

1 # Build image agent
2 docker build -t danielcrsth0/agent:1.1 .

```

---

#### Kode Sumber 3.19: Perintah untuk Build dan Menjalankan Agent

---

```

1 # Push image agent
2 docker push danielcrsth0/agent:1.1

```

---

#### Kode Sumber 3.20: Perintah untuk Push Image Agent

Untuk menjalankan agent pada setiap node, digunakan perintah `docker run` berikut. Perintah ini menjalankan container agent sebagai daemon ('-d') dengan mode jaringan '-net=host' agar agent dapat mengakses IP dan port host secara langsung. Variabel lingkungan `DISCOVERY_URL` digunakan untuk menentukan endpoint `/register-node` pada Discovery API pusat. Selain itu, file `/var/run/docker.sock` dimount ke dalam container agar agent dapat mengakses informasi container yang sedang berjalan di node tersebut. Untuk node yang memiliki GPU, tambahan opsi `--gpus=all` digunakan agar container agent dapat membaca informasi GPU menggunakan *library* seperti `nvidia-smi` atau `gpustat`.

---

```

1
2 # Menjalankan agent di setiap node
3 docker run --name agent -d \
4     --net=host \
5     -e DISCOVERY_URL=http://10.33.17.30:18000:15002/register-node \
6     -v /var/run/docker.sock:/var/run/docker.sock \
7     danielcrsth0/agent:1.1
8
9 # Tambahkan "--gpus all" di node yang memiliki GPU
10 docker run --name agent -d \
11     --net=host \
12     -e DISCOVERY_URL=http://10.33.17.30:15002/register-node \
13     -v /var/run/docker.sock:/var/run/docker.sock \
14     --gpus all \
15     danielcrsth0/agent:1.1

```

---

#### Kode Sumber 3.21: Perintah untuk Menjalankan Agent di setiap Node

Perintah pada Kode Sumber 3.21 digunakan untuk men-*deploy* service agent sebagai *container* Docker di setiap *node* dalam kluster. Variasi perintah disesuaikan dengan jenis *node*, baik dengan atau tanpa GPU. Penjelasan mengenai konfigurasi parameter perintah ini telah dibahas pada bagian sebelumnya.

Selanjutnya untuk melihat informasi apa saja yang dilaporkan *agent* ke *service discovery* bisa menggunakan perintah **docker logs -f agent**, dimana bisa dilihat *node agent* akan melaporkan informasi ke *service discovery* melalui URL **http://10.33.17.30/register-node**. Informasi yang dilaporkan adalah jumlah *container* yang sedang aktif di *node* atau komputer tersebut, *hostname* dari *node* tersebut, alamat IP, informasi CPU (jumlah *cores* yang dimiliki dan persentase penggunaan saat ini), informasi memory (ukuran memori yang dimiliki dan persentase yang digunakan saat ini) dan informasi GPU jika *node* tersebut memiliki GPU. Untuk contoh dari log *container agent* dapat dilihat pada gambar di bawah ini:



```
daniel@rpl:~/ray-jupyterhub-ml/service-agent$ docker logs agent -f
[AGENT] Starting node registration agent...
[AGENT] Target URL: http://10.33.17.30:15002/register-node
[DEBUG] adding node...
[DEBUG] Menggunakan alamat IP 10.21.73.139 yang dispesifikasi pada interface k3s-br0
[DEBUG] Container Summary: Total=3, JupyterLab=0, Ray=0
[DEBUG] Send Info: {'hostname': 'rpl', 'ip': '10.21.73.139', 'cpu_cores': 24, 'gpu_info': [{'name': 'NVIDIA GeForce RTX 3080 Ti', 'index': 0, 'uuid': 'GPU-56c3e796-124e-f059-16a8-f9be2b254ce0', 'memory_total_mb': 12288, 'memory_used_mb': 10837, 'memory_util_percent': 88.19, 'utilization_gpu_percent': 0, 'temperature_gpu': 41}], 'has_gpu': True, 'ram_gb': 67.11, 'max_containers': 10, 'is_active': True, 'cpu_usage_percent': 4.5, 'memory_usage_percent': 19.3, 'disk_usage_percent': 51.5, 'active_jupyterlab': 0, 'active_ray': 0, 'total_containers': 3, 'last_updated': '2025-07-09T06:24:45.869276Z'}
[AGENT] Registered: rpl (10.21.73.139) - 200
```

Gambar 3.5: Log dari container agent ketika melakukan registrasi node

### 3.2.3 Implementasi JupyterHub dan Jupyter Enterprise Gateway

JupyterHub bertindak sebagai sistem autentikasi dan pengelola sesi pengguna. Setiap pengguna dapat memulai server JupyterLab pribadi yang dijalankan sebagai *container* terisolasi. Dengan bantuan *spawner* khusus yang terintegrasi dengan Discovery Service, JupyterHub akan secara otomatis memilih *node* dengan *resource* teringan. *Spawner* ini juga bertugas melakukan konfigurasi *container* secara otomatis, termasuk *setting* IP, *port*, dan *image* sesuai kebutuhan pengguna.

Untuk mendukung eksekusi *kernel* secara terdistribusi, sistem ini mengintegrasikan Jupyter Enterprise Gateway (JEG) sebagai komponen perantara antara JupyterHub dan *node* tempat *kernel* dijalankan. JEG dikonfigurasi untuk menerima permintaan peluncuran *kernel* dari *spawner*, kemudian menjalankannya pada *node* yang dipilih melalui *process proxy* yang sesuai. Konfigurasi *kernel* dihasilkan secara dinamis oleh sistem dan mencakup informasi seperti perintah eksekusi, alamat IP *node* tujuan, serta parameter koneksi yang diperlukan. Dengan mekanisme ini, *kernel* tidak dijalankan langsung di server utama, melainkan di *node* lain yang telah ditentukan, sehingga mendukung fleksibilitas dalam pengelolaan sumber daya.

Sub-bab berikut ini merinci konfigurasi dasar yang menjadi fondasi dari platform JupyterHub yang dikembangkan. Konfigurasi ini mencakup manajemen autentikasi pengguna, pengaturan inti dari proses Hub, konfigurasi proxy sebagai gerbang utama, serta mekanisme hook untuk pemantauan. Pendekatan modular digunakan di mana setiap aspek fungsional diisolasi dalam filenya sendiri untuk kemudahan pengelolaan.



### 3.2.3.1 Arsitektur Aplikasi

Proyek JupyterHub ini dibangun dengan struktur modular yang memisahkan konfigurasi, spawner, dan form HTML dalam direktori berbeda. Hal ini memudahkan pemeliharaan dan pengembangan fitur baru.

Tabel 3.6: Struktur Direktori Proyek JupyterHub

Nama File/Folder	Deskripsi
jupyterhub_config.py	Titik masuk konfigurasi utama JupyterHub.
config/	Konfigurasi modular auth, spawner, hooks, dan proxy.
spawner/	Implementasi custom MultiNodeSpawner.
form/	Template HTML dan JS untuk interface pemilihan node.
singleuser/	Dockerfile dan skrip build image JupyterLab.
docker-compose.yml	Orkestrasi layanan JupyterHub dan reverse proxy.

### 3.2.3.2 Inisialisasi Konfigurasi dan Komponen

File `jupyterhub_config.py` berperan sebagai titik masuk konfigurasi yang memanggil fungsi-fungsi konfigurasi modular dari direktori `config/`. Ini termasuk konfigurasi environment, autentikasi, proxy, spawner, serta hook yang diperlukan saat spawn dan terminasi container.

```
1 // Impor modul konfigurasi
2 IMPORT load_environment FROM config.env
3 IMPORT configure_hub FROM config.hub
4 IMPORT configure_spawner FROM config.spawner
5 IMPORT configure_proxy FROM config.proxy
6 IMPORT configure_auth FROM config.auth
7 IMPORT attach_hooks FROM config.hooks
8
9 // Panggil setiap fungsi untuk melakukan konfigurasi
10 CALL load_environment WITH c
11 CALL configure_hub WITH c
12 CALL configure_spawner WITH c
13 CALL configure_proxy WITH c
14 CALL configure_auth WITH c
15 CALL attach_hooks
```

Kode Sumber 3.22: Kode Semu file Jupyterhub Config

### 3.2.3.3 Autentikasi dan Manajemen Pengguna

File `config/auth.py` berisi konfigurasi manajemen pengguna, sistem ini memanfaatkan **NativeAuthenticator**, sebuah mekanisme otentikasi bawaan dari JupyterHub. Pendekatan ini dipilih karena kemudahannya, di mana pengguna dapat mendaftarkan akunnya secara mandiri tanpa memerlukan integrasi ke sistem autentikasi eksternal. Konfigurasi utama pada `config/auth.py` mencakup:

- Opsi `open_signup` diaktifkan agar pengguna baru dapat membuat akun terlebih dahulu.

- Menerapkan kebijakan keamanan dasar dengan panjang kata sandi minimal 8 karakter.

---

```

1 PROCEDURE Configure_Authentication:
2     // Menetapkan NativeAuthenticator sebagai kelas otentikasi utama
3     SET AuthenticatorClass TO "NativeAuthenticator"
4
5     // Mengizinkan pengguna baru untuk mendaftar secara mandiri
6     ENABLE SelfRegistration
7
8     // Menetapkan kebijakan keamanan dasar untuk kata sandi
9     SET MinimumPasswordLength TO 8
10
11    // Mengambil daftar admin dari environment variable untuk ←
12    // fleksibilitas
13    DEFINE AdminUsers FROM EnvironmentVariable("JUPYTERHUB_ADMIN")
14 END PROCEDURE

```

---

Kode Sumber 3.23: Kode Semu untuk Konfigurasi Autentikasi

### 3.2.3.4 Konfigurasi Inti dan Layanan Hub

File **config/hub.py** berisi konfigurasi untuk proses inti Hub, termasuk konektivitas jaringan, integrasi dengan basis data sebagai tempat penyimpanan yang persistent, dan layanan **Idle Culler** yang digunakan untuk mematikan server JupyterLab yang tidak aktif.

---

```

1 PROCEDURE Configure_Core_Hub:
2     // --- Pengaturan Jaringan untuk Lingkungan Docker ---
3     BIND HubInternalURL TO "http://0.0.0.0:18000"
4     SET HubInternalHostname TO "hub"
5
6     // --- Pengaturan Basis Data ---
7     // Menggunakan PostgreSQL sebagai database
8     READ DatabaseConfig FROM EnvironmentVariables (Host, Port, User, Pass←
9     , DBName)
10    SET DatabaseURL TO Format("postgresql://{User}:{Pass}@{Host}:{Port}/{←
11    DBName}")
12
13    ENABLE ShutdownServersOnUserLogout
14    ENABLE CleanupServersOnHubRestart
15
16    // Mengonfigurasi layanan untuk mematikan container JupyterLab yang ←
17    // tidak aktif
18    DEFINE Service "IdleCuller" WITH Command:
19        "shutdown_idle_servers_after(600 seconds)"
20 END PROCEDURE

```

---

Kode Sumber 3.24: Kode Semu untuk Konfigurasi Inti Hub

### 3.2.3.5 Konfigurasi Proxy

Dalam arsitektur sistem ini, proxy dirancang untuk berjalan sebagai layanan (kontainer) yang independen, terpisah dari kontainer Hub. Proxy di JupyterHub adalah jembatan utama yang mengarahkan permintaan user ke tempat yang tepat, menjaga isolasi, keamanan, dan menyederhanakan akses multi-user dalam sistem yang kompleks.

---

```

1 PROCEDURE Configure_Proxy:
2     // Mengindikasikan bahwa proxy adalah layanan eksternal
3     INDICATE ProxyIsManagedExternally
4
5     // Menetapkan alamat API internal untuk berkomunikasi dengan proxy
6     SET Proxy_API_URL TO "http://proxy:8001"
7
8     // Menggunakan token dari environment variable untuk komunikasi yang ↵
9     SET Proxy_Auth-Token FROM EnvironmentVariable("CONFIGPROXY_AUTH_TOKEN↵
10 END PROCEDURE

```

---

Kode Sumber 3.25: Kode Semu untuk Konfigurasi Proxy

### 3.2.3.6 Integrasi Multi-Node dan Spawner Khusus

Untuk mencapai tujuan utama penelitian ini—yaitu alokasi sumber daya yang dinamis dan terdistribusi, sistem ini mengimplementasikan sebuah spawner kustom yang disebut **MultiNodeSpawner**. Komponen ini merupakan turunan dari **DockerSpawner** bawaan JupyterHub, namun dengan modifikasi signifikan untuk dua tujuan utama:

1. Menjalankan atau meluncurkan environment JupyterLab pada node paling optimal yang dipilih melalui komunikasi dengan Discovery Service.
2. Mengonfigurasi environment tersebut secara dinamis agar dapat memanfaatkan node-node lain dalam kluster untuk eksekusi kode melalui Jupyter Enterprise Gateway (JEG).

Implementasi spawner sendiri terbagi dalam dua kelas utama: **MultiNodeSpawner** sebagai logika dasar dan **PatchedMultiNodeSpawner** sebagai lapisan perbaikan untuk memastikan integrasi yang mulus.

### 3.2.3.7 Implementasi MultiNodeSpawner

MultiNodeSpawner diimplementasikan sebagai subclass dari DockerSpawner dan bertugas menjalankan container JupyterLab pada node yang dipilih berdasarkan informasi dari Discovery Service. Spawner ini mengakses endpoint `/select-nodes` dengan parameter berupa `profile_id` dan `user_id`, lalu menerima daftar node dengan skor beban terbaik.

Jika profil pengguna membutuhkan lebih dari satu node, maka Spawner akan menyimpan daftar node tersebut dalam atribut `selected_nodes` untuk nantinya menjalankan Container JupyterLab.

---

```

1 PROCEDURE start():
2     // 1. Membaca input dari form pengguna
3     Parse_User_Selections()
4
5     // 2. Menentukan di mana server utama JupyterLab akan berjalan
6     SELECT primary_node FROM user_selected_nodes
7
8     // 3. Mengarahkan Spawner untuk beroperasi pada node utama
9     CONNECT_To_Docker_Daemon(primary_node.ip)
10

```

---

```

11 // 4. Membuat file konfigurasi kernel untuk JEG
12 Generate_And_Write_JEG_Kernels(user_selected_nodes)
13
14 // 5. Meluncurkan server JupyterLab dengan konfigurasi JEG
15 Launch_JupyterLab_Container()
16 END PROCEDURE
17
18 PROCEDURE stop():
19 // Menghentikan kontainer server JupyterLab
20 CALL parent.stop()
21
22 // Membersihkan file-file konfigurasi kernel yang dinamis
23 CLEANUP_JEG_Kernel_Files()
24 END PROCEDURE

```

---

Kode Sumber 3.26: Kode Semu untuk Logika MultiNodeSpawner

### 3.2.3.8 *Generate* Kernel Dinamis untuk Integrasi dengan JEG

Salah fungsi dari MultiNodeSpawner adalah kemampuannya secara dinamis membuat file konfigurasi kernel (*kernelspecs*) untuk setiap node yang dipilih oleh pengguna. Mekanisme ini memungkinkan Jupyter Enterprise Gateway (JEG) untuk mengetahui cara meluncurkan kernel Python di berbagai *node* dalam kluster.

Logika ini diimplementasikan dalam metode `_generate_kernelspecs_config` dan `_write_kernel`. Untuk setiap node yang dipilih, sistem akan membuat sebuah file **kernel.json**. File ini tidak berisi kode Python, melainkan sebuah resep (konfigurasi) bagi JEG yang memberitahukan:

1. Perintah **docker run** yang harus dijalankan JEG di node target untuk memulai sebuah kernel di dalam kontainer.
2. Menggunakan **DistributedProcessProxy** dari JEG untuk mengelola komunikasi antara JupyterLab dan kernel yang berjalan di lokasi terpencil.
3. Informasi seperti nama kernel yang akan tampil di antarmuka JupyterLab (misalnya, *"Python 3 on node-A"*).

---

```

1 PROCEDURE Generate_And_Write_JEG_Kernels(selected_nodes):
2 // Membuat direktori bersama untuk menyimpan semua file kernel
3 CREATE_DIRECTORY "/srv/jupyterhub/kernels"
4
5 // Iterasi untuk setiap node yang dipilih pengguna
6 FOR EACH node IN selected_nodes:
7 // Menyiapkan perintah untuk dieksekusi oleh JEG di node remote
8 DEFINE kernel_command AS "docker run --network=host <kernel_image↵
> ..."
9
10 DEFINE kernel_spec AS {
11 display_name: "Python on {node.hostname}",
12 language: "python",
13 process_proxy_class: "DistributedProcessProxy",
14 argv: kernel_command,
15 env: { NODE_IP: node.ip }
16 }
17 WRITE kernel_spec TO "/srv/jupyterhub/kernels/{node.hostname}/↵
kernel.json"

```

```

17     END FOR
18 END PROCEDURE

```

---

Kode Sumber 3.27: Kode Semu untuk Generasi Konfigurasi Kernel JEG

### 3.2.3.9 Implementasi PatchedMultiNodeSpawner

File `multinode.py` berisi kelas `PatchedMultiNodeSpawner` yang mewarisi `MultiNodeSpawner` dan menambahkan perbaikan berikut:

- Perbaikan properti `server_url` agar selalu valid
- Sinkronisasi nilai IP dan port container ke variabel internal
- Penyesuaian konfigurasi URL dan argumen server Jupyter

Listing 3.29 menunjukkan implementasi override URL pada `PatchedMultiNodeSpawner`:

---

```

1 // Properti untuk membangun alamat dasar server (IP dan port)
2 PROPERTY server_url:
3     IF ip AND port are available THEN
4         RETURN "http://{ip}:{port}"
5     ELSE
6         RETURN empty_string
7     END IF
8 END PROPERTY
9 // Properti untuk membangun URL lengkap yang akan diakses pengguna
10 PROPERTY url:
11     // Mulai dengan URL dasar dari server_url
12     SET base_url TO self.server_url
13     // Jika ada path aplikasi default (misal: "/lab"), tambahkan
14     IF a default_url path exists THEN
15         APPEND default_url path to base_url
16     END IF
17     // Kembalikan URL yang sudah lengkap
18     RETURN base_url
19 END PROPERTY

```

---

Kode Sumber 3.28: Kode Semu untuk Properti URL pada Spawner

### 3.2.3.10 Konfigurasi Jupyter Enterprise Gateway

Jupyter Enterprise Gateway (JEG) dikonfigurasi untuk menjalankan kernel secara remote melalui koneksi SSH tanpa sandi. Konfigurasi ini mencakup pengaturan waktu koneksi, daftar node yang diizinkan, serta algoritma distribusi beban untuk pemilihan node. Penggunaan algoritma **Least Connection** memungkinkan distribusi kernel secara adaptif berdasarkan jumlah koneksi aktif, dan dinilai lebih responsif dibanding round-robin pada kondisi beban yang tidak merata. File konfigurasi juga menyertakan pengaturan rentang port komunikasi dan format log. Seluruh konfigurasi ini dituliskan pada *file* `jeg_config.py`.

---

```

1 PROCEDURE Configure_Jupyter_Enterprise_Gateway:
2     // Menetapkan alamat untuk menerima response dari kernel
3     SET ResponseAddress TO "0.0.0.0:8877"
4
5     // Mengatur timeout untuk kernel yang tidak aktif

```

```

6  SET IdleTimeout TO 3600 // dalam detik
7  SET CullInterval TO 600 // dalam detik
8
9  // Mengatur batas waktu saat inisialisasi kernel remote
10 SET SocketTimeout TO 5.0
11 SET PrepareTimeout TO 120.0
12
13 // Menetapkan daftar node yang diizinkan
14 DEFINE RemoteHosts AS [
15     "10.21.73.107",
16     "10.21.73.125",
17     "10.21.73.139"
18 ]
19
20 // Mengatur strategi distribusi kernel
21 SET LoadBalancingAlgorithm TO "least-connection"
22
23 // Menentukan rentang port yang digunakan kernel
24 SET KernelPortRange TO "40000..50000"
25
26 // Nonaktifkan pengecekan host key SSH
27 ENABLE DisableHostKeyChecking
28
29
30 // Konfigurasi format log aplikasi
31 SET LogFormat TO "[% (levelname)1.1s %(asctime)s.%(msecs).03d %(name)s] ←
    %(message%)"
32
33 END PROCEDURE

```

---

Kode Sumber 3.29: File Konfigurasi JEG

### 3.2.3.11 Deployment JupyterHub dan JEG dengan Docker Compose

Untuk memudahkan proses deployment di berbagai lingkungan, Layanan JupyterHub dan JEG dikemas dalam *image* menggunakan Docker. Selanjutnya diatur dengan Docker Compose untuk menjalankan seluruh komponen (Proxy, Hub & JEG) secara terorkestrasi.

Dockerfile ini bertanggung jawab untuk membangun *image* Docker kustom untuk JupyterHub. Image dasar atau *base image* yang digunakan adalah **quay.io/jupyter/base-notebook:hub-5.3.0**.

Beberapa konfigurasi tambahan dilakukan, antara lain instalasi dependensi, penambahan konfigurasi `jupyterhub_config.py`, direktori form untuk antarmuka `anspawner`. User yang digunakan adalah `jovyan` (UID 1000) untuk alasan kompatibilitas dengan ekosistem Jupyter.

---

```

1 FROM base-notebook:hub-5.3.0
2 INSTALL system dependencies
3 COPY requirements.txt dan install Python packages
4 COPY konfigurasi JupyterHub:
5     - jupyterhub_config.py
6     - direktori form, config, spawner
7 SET WORKDIR /srv/jupyterhub
8 EXPOSE port 18000
9 CMD jupyterhub --config /srv/jupyterhub/jupyterhub\_config.py

```

---

Kode Sumber 3.30: Dockerfile JupyterHub

Image JEG dibangun dari Docker *image* **elyra/enterprise-gateway:3.2.3**. *Image* ini dikustomisasi untuk mendukung komunikasi SSH ke *node remote* dan mendukung konfigurasi tambahan seperti **launch\_ipykernel.py** dan **jeg\_config.py**.

---

```
1 FROM elyra/enterprise-gateway:3.2.3
2 INSTALL ssh client, netcat, docker, cryptography
3 COPY konfigurasi:
4     - jeg_config.py TO /etc/jupyter/
5     - entrypoint.sh TO /usr/local/bin/
6     - launch_ipykernel.py TO /usr/local/share/jupyter/kernels/
7 SET PERMISSION pada SSH keys
8 SET ENV (EG\_PORT, EG\_AUTH\_TOKEN, EG\_RESPONSE\_IP, EG\_REMOTE\_USER, dst↔
9 )
9 ENTRYPOINT ["entrypoint.sh"]
```

---

Kode Sumber 3.31: Dockerfile JEG

Orkestrasi layanan dilakukan menggunakan Docker Compose, dengan berkas `docker-compose.yml` sebagai pusat definisi deployment multi-container. Berkas ini mendeskripsikan beberapa layanan utama yang saling terintegrasi. Layanan pertama adalah JupyterHub (`hub`), yang bertugas untuk melakukan autentikasi pengguna serta melakukan spawning container sesuai konfigurasi yang diberikan. Selanjutnya, terdapat layanan Jupyter Enterprise Gateway (`jeg`) yang berfungsi untuk menjalankan kernel secara terdistribusi pada *node-node remote*. Layanan ketiga adalah `proxy`, yang menggunakan `configurable-http-proxy` untuk menangani routing lalu lintas HTTP antara pengguna, `Hub`, dan server JupyterLab yang dijalankan. Dengan struktur ini, tiap komponen berjalan secara terisolasi namun tetap dapat berkomunikasi melalui jaringan internal Docker.

---

```
1 services:
2   proxy:
3     image: configurable-http-proxy
4     ports: 18000:8000
5     depends_on: hub
6
7   hub:
8     build: ./hub
9     image: jupyterhub:1.1
10    volumes:
11      - /var/run/docker.sock
12      - /opt/jupyterhub/kernels:/srv/jupyterhub/kernels
13    environment:
14      - JUPYTER_GATEWAY_URL=http://jeg:8889
15    expose: 8081
16
17   jeg:
18     build: ./jeg
19     image: gateway:1.1
20     ports: 8889:8889, 8877:8877
21     volumes:
22       - ./launch_ipykernel.py:/usr/local/share/jupyter/kernels/...
23     environment:
24       - EG_REMOTE_USER
25       - EG_PORT
```

---

Kode Sumber 3.32: Orkestrasi beberapa layanan dengan Docker Compose

### 3.3 Peralatan Pendukung

Perangkat yang digunakan dalam pengerjaan tugas akhir ini terdiri dari empat unit komputer (*node*) yang saling terhubung dalam satu jaringan lokal. Keempat node ini digunakan untuk mendistribusikan beban komputasi serta mendukung pengembangan dan pengujian sistem secara menyeluruh. Satu node berfungsi sebagai control node, yang menjalankan layanan utama seperti JupyterHub, Jupyter Enterprise Gateway (JEG), dan proxy service menggunakan Docker Compose. Node ini juga difungsikan sebagai development environment untuk menulis kode, mengelola kontainer, serta melakukan konfigurasi sistem. Tiga node lainnya bertindak sebagai worker node dan digunakan untuk menjalankan server JupyterLab serta kernel secara terdistribusi. Salah satu dari ketiga worker node tersebut, yaitu node RPL, dilengkapi dengan unit pemrosesan grafis (GPU) dan secara khusus digunakan untuk mengeksekusi workload berbasis pembelajaran mesin (*machine learning*) yang membutuhkan akselerasi komputasi. Sistem ini dibangun dengan pendekatan multi-node orchestration agar mampu mengelola lingkungan komputasi terdistribusi dan multi-user secara efisien. Spesifikasi lengkap dari masing-masing node dapat dilihat pada di bawah ini.



Tabel 3.7: Spesifikasi Peralatan Pendukung

No.	Komponen	Spesifikasi
1	<b>Laptop</b>	
	<i>Brand</i>	Asus
	<i>Operating System</i>	Ubuntu 22.04 LTS
	<i>Processor</i>	AMD Ryzen 3
	<i>GPU</i>	AMD Radeon vega 3 graphics
	<i>CPU Cores</i>	4 Cores
	<i>Memory</i>	18 GB
	<i>Storage</i>	512 GB
2	<b>Komputer RPL</b>	
	<i>Brand</i>	Asus
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>Processor</i>	Intel Core i9 Gen 12
	<i>GPU</i>	NVIDIA GeForce RTX 3080 Ti
	<i>CPU Cores</i>	24 Cores
	<i>Memory</i>	64 GB
	<i>Storage</i>	723 GB
3	<b>Komputer RPL 1</b>	
	<i>Brand</i>	Asus
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>Processor</i>	Intel Core i7 Gen 12
	<i>CPU Cores</i>	20 Cores
	<i>Memory</i>	16 GB
	<i>Storage</i>	1 TB
4	<b>Komputer RPL 2</b>	
	<i>Brand</i>	Asus
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>Processor</i>	12th Gen Intel i7-12900K
	<i>CPU Cores</i>	20 Cores
	<i>Memory</i>	32 GB
	<i>Storage</i>	1 TB

Selain perangkat keras, terdapat juga perangkat lunak pendukung seperti berikut.

Tabel 3.8: Daftar Perangkat Lunak Pendukung

Nama Perangkat Lunak	Versi	Keterangan
Docker Engine	28.2.2	Digunakan untuk menjalankan container JupyterLab secara terisolasi. Memungkinkan lingkungan komputasi tiap pengguna berjalan secara independen dan mudah didistribusikan ke berbagai node.
Docker Compose	2.36.2	Membantu mendefinisikan dan mengatur layanan multi-container JupyterHub dan Service Discovery dalam satu berkas konfigurasi. Memudahkan manajemen dan replikasi layanan.
Python	3.11	Bahasa pemrograman utama yang digunakan untuk seluruh komponen sistem, seperti konfigurasi JupyterHub, pengembangan REST API, serta skrip monitoring untuk <i>service agent</i> .
PostgreSQL	14	Basis data relasional yang digunakan untuk menyimpan data historis seperti riwayat pemilihan node, profil pengguna, dan metrik performa dari setiap node.
Redis	7.0	Database key-value in-memory yang digunakan untuk menyimpan status sistem (CPU, RAM, GPU) dan log aktivitas pengguna secara real-time.
Flask	3.1.0	Framework web Python yang digunakan untuk membangun <i>service discovery</i> berupa REST API yang menerima dan menyediakan data status node.
JupyterHub	5.3.0	Menangani autentikasi pengguna serta spawn <i>container</i> JupyterLab ke node terpilih berdasarkan data dari <i>service discovery</i> .
Jupyter Enterprise Gateway	3.2.3	Jupyter Enterprise Gateway adalah ekstensi dari Jupyter yang memungkinkan eksekusi <i>kernel</i> Jupyter di lingkungan terdistribusi. Dengan JEG, <i>kernel</i> tidak lagi berjalan di server yang sama dengan JupyterHub, melainkan dapat dijalankan di <i>node</i> lain (remote host) secara terpisah, cocok untuk arsitektur <i>multi-node</i> .
Locust	2.37.11	Locust adalah alat uji beban ( <i>load testing</i> ) berbasis Python yang digunakan untuk mensimulasikan banyak pengguna secara bersamaan. Dalam penelitian ini, Locust digunakan untuk menguji kemampuan sistem dalam menangani permintaan multi-user concurrent terhadap JupyterHub.

*[Halaman ini sengaja dikosongkan]*

## BAB IV

# HASIL dan PEMBAHASAN

Bab ini membahas proses pengujian dan hasil analisis terhadap sistem yang telah dibangun. Tujuan utama dari pengujian ini adalah untuk mengevaluasi kinerja Service Discovery dalam memilih node yang optimal untuk menjalankan container JupyterLab, serta memastikan bahwa integrasi antar komponen (JupyterHub, Discovery API, Agent, dan Docker) berjalan sesuai ekspektasi.

### 4.1 Hasil Implementasi

#### 4.1.1 Implementasi Service Agent

Agent akan membaca informasi sistem seperti jumlah CPU, kapasitas RAM, penggunaan GPU (jika tersedia), dan jumlah *container* yang berjalan. Informasi tersebut kemudian dikemas dalam bentuk *payload* JSON dan dikirimkan melalui HTTP POST ke endpoint `/register-node` pada Discovery API.

Berikut merupakan contoh keluaran log saat proses registrasi node berhasil:

```
1 [AGENT] Registered: rpl (10.21.73.139)      200
2 [DEBUG] adding node...
3 [DEBUG] Menggunakan alamat IP 10.21.73.139 yang dispesifikan pada ↵
   interface k3s-br0
4 [DEBUG] Container Summary: Total=9, JupyterLab=5, Ray=0
5 [DEBUG] Send Info: {
6   'hostname': 'rpl',
7   'ip': '10.21.73.139',
8   'cpu_cores': 24,
9   'gpu_info': [{
10    'name': 'NVIDIA GeForce RTX 3080 Ti',
11    'index': 0,
12    'uuid': 'GPU-56c3e796-124e-f059-16a8-f9be2b254ce0',
13    'memory_total_mb': 12288,
14    'memory_used_mb': 10547,
15    'memory_util_percent': 85.83,
16    'utilization_gpu_percent': 0,
17    'temperature_gpu': 31
18  }],
19   'has_gpu': True,
20   'ram_gb': 67.11,
21   'max_containers': 10,
22   'is_active': True,
23   'cpu_usage_percent': 0.9,
24   'memory_usage_percent': 15.3,
25   'disk_usage_percent': 57.0,
26   'active_jupyterlab': 5,
27   'total_containers': 9,
28   'last_updated': '2025-07-10T08:52:40.819686Z'
29 }
```

---

#### Kode Sumber 4.1: Contoh log registrasi node

Dari log di atas, dapat dilihat bahwa node dengan hostname `rpl` memiliki spesifikasi 24 *cores* CPU dan RAM sebesar 67.11 GB. Node ini juga dilengkapi dengan GPU dan menjalankan total 9 container, di mana 5 di antaranya merupakan *container* JupyterLab.

Data ini kemudian disimpan oleh Discovery API ke dalam penyimpanan Redis dengan, monitoring status node secara real-time.

### 4.1.2 Implementasi Service Discovery

Discovery API merupakan komponen utama yang bertanggung jawab terhadap penyimpanan data status node serta penentuan node terbaik berdasarkan beban kerja dan sumber daya yang tersedia. Layanan ini diimplementasikan menggunakan framework Flask dan dijalankan sebagai API.

Tabel 4.1: Daftar Endpoint REST API pada Discovery Service

Metode	Endpoint	Fungsi
GET	/health-check	Mengecek status koneksi layanan, termasuk status Redis dan PostgreSQL.
POST	/register-node	Menerima informasi node dari Agent dan menyimpan status terbaru ke Redis serta basis data.
GET	/available-nodes	Mengambil daftar node aktif beserta skor beban terkini.
POST	/select-nodes	Memilih sejumlah node berdasarkan algoritma load balancing tertentu.
GET	/all-nodes	Menampilkan semua node yang pernah terdaftar, termasuk node yang tidak aktif.
GET	/profiles	Menampilkan daftar seluruh profil user yang tersedia.
POST	/profiles	Menambahkan profil baru ke sistem.
PUT	/profiles/<id>	Memperbarui konfigurasi profil berdasarkan ID.
DELETE	/profiles/<id>	Menghapus profil dari sistem berdasarkan ID.

Struktur utama dari Discovery API terdiri dari beberapa komponen utama:

- **Endpoint REST API** seperti `/register-node`, `/available-nodes`, dan `/select-nodes`.
- **Penyimpanan sementara (Redis)** untuk menyimpan informasi node secara real-time dengan TTL.
- **Penyimpanan Persisten (PostgreSQL)** untuk pencatatan log pemilihan node dan statistik.
- **Algoritma pemilihan node** berbasis skor dengan parameter CPU, RAM, GPU, dan jumlah *container* yang aktif.

Endpoint `/available-nodes` digunakan oleh komponen eksternal seperti JupyterHub untuk memperoleh daftar node yang aktif beserta hasil seleksi node terbaik. Berikut merupakan contoh payload yang dikembalikan oleh endpoint tersebut:

```
1 {
2   "all_available_nodes": [
3     {
4       "hostname": "rpl",
5       "ip": "10.21.73.139",
6       "cpu_cores": 24,
7       "ram_gb": 67.11,
8       "cpu_usage_percent": 0.8,
9       "memory_usage_percent": 15.7,
10      "disk_usage_percent": 57.1,
11      "total_containers": 9,
12      "active_jupyterlab": 5,
13      "load_score": 13.2
14    },
15    {
16      "hostname": "rpl-02",
17      "ip": "10.21.73.125",
18      "cpu_cores": 20,
19      "ram_gb": 33.38,
20      "cpu_usage_percent": 0.5,
21      "memory_usage_percent": 35.8,
22      "disk_usage_percent": 78.6,
23      "total_containers": 2,
24      "active_jupyterlab": 1,
25      "load_score": 29.04
26    },
27    {
28      "hostname": "chrstdan",
29      "ip": "10.125.180.220",
30      "cpu_cores": 4,
31      "ram_gb": 18.79,
32      "cpu_usage_percent": 13.7,
33      "memory_usage_percent": 69.5,
34      "disk_usage_percent": 97.6,
35      "total_containers": 0,
36      "active_jupyterlab": 0,
37      "load_score": 66.56
38    }
39  ],
40  "load_balancing": {
41    "algorithm": "round_robin",
42    "requested_count": 1,
43    "round_robin_counter": 5,
44    "selected_count": 1
45  },
46  "selected_nodes": [
47    {
48      "hostname": "rpl-02",
49      "ip": "10.21.73.125",
50      "cpu_cores": 20,
51      "ram_gb": 33.38,
52      "cpu_usage_percent": 0.5,
53      "memory_usage_percent": 35.8,
```

```

54     "disk_usage_percent": 78.6,
55     "total_containers": 2,
56     "active_jupyterlab": 1,
57     "load_score": 29.04
58   }
59 ],
60 "total_available_nodes": 3
61 }

```

Kode Sumber 4.2: Contoh respons Discovery API pada endpoint /available-nodes

Dari data di atas, diketahui terdapat tiga node aktif: `rpl`, `rpl-02`, dan `chrstdan`. Node `rpl` memiliki kapasitas tertinggi (24 CPU, 67 GB RAM) dengan beban rendah (`load_score` 13.2), sementara node `chrstdan` menunjukkan beban yang sangat tinggi dengan `load_score` mencapai 66.56 dan penggunaan disk sebesar 97.6%.

Node yang terpilih dalam contoh ini adalah `rpl-02`, meskipun `rpl` memiliki skor beban yang lebih rendah. Hal ini terjadi karena sistem menerapkan algoritma *round robin*, di mana pemilihan node dilakukan secara bergilir dengan tetap mempertimbangkan status aktif dan batas maksimum *container*. Nilai `round_robin_counter` menunjukkan bahwa iterasi ke-5 menghasilkan pilihan terhadap node kedua dalam daftar.




```

docker exec -it redis redis-cli -a redis@pass -p 16379
127.0.0.1:16379> KEYS *
1) "node:chrstdan:ip"
2) "node:docker-desktop:info"
3) "node:docker-desktop:ip"
4) "node:rpl:ip"
5) "node:rpl:info"
6) "node:rpl-02:ip"
7) "node:rpl-02:info"
8) "node:chrstdan:info"
127.0.0.1:16379> GET node:rpl:info
"{\"hostname\": \"rpl\", \"ip\": \"10.21.73.139\", \"cpu_cores\": 24, \"gpu_in
fo\": [], \"has_gpu\": false, \"ram_gb\": 67.11, \"max_containers\": 10, \"is
active\": true, \"cpu_usage_percent\": 0.8, \"memory_usage_percent\": 15.7, \"
disk_usage_percent\": 57.0, \"active_jupyterlab\": 5, \"active_ray\": 0, \"tot
al_containers\": 9, \"last_updated\": \"2025-07-10T10:32:57.377505Z\"}"
127.0.0.1:16379> GET node:rpl:ip
"10.21.73.139"
127.0.0.1:16379>

```

Gambar 4.1: Informasi dari setiap node berhasil disimpan ke Redis



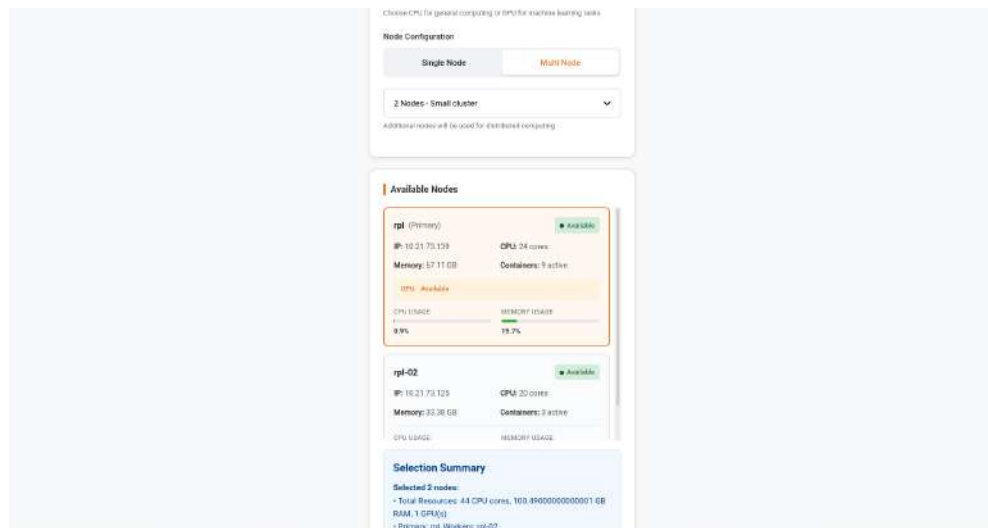
id	hostname	ip	cpu_cores	ram_gb	has_gpu	gpu_info	is_active	max_container	created_at	updated_at
201	rpl	10.21.73.139	24	67.11	FALSE	[]	TRUE	10	2025-07-08 20:34:14.2...	2025-07-10 10:23:13.4...
25	rpl-02	10.21.73.125	20	33.38	FALSE	[]	TRUE	10	2025-07-08 16:59:46.5...	2025-07-10 10:23:17.1...
3	chrstdan	10.125.180...	4	18.79	FALSE	[]	TRUE	10	2025-06-22 00:32:27.1...	2025-07-10 10:23:19.4...
1	worker1	192.168.122...	2	4.1	FALSE	[]	FALSE	10	2025-06-21 14:21:28.1...	2025-07-08 18:10:43.7...
35	rpl-1	10.21.73.107	20	16.5	FALSE	[]	FALSE	10	2025-07-08 19:52:57.4...	2025-07-08 20:50:04.8...
2	worker2	192.168.122...	2	4.1	FALSE	[]	FALSE	10	2025-06-21 14:21:28.1...	2025-07-08 16:36:37.1...

Gambar 4.2: Informasi dari setiap node berhasil disimpan ke PostgreSQL di tabel `nodes`

### 4.1.3 Hasil Implementasi JupyterHub

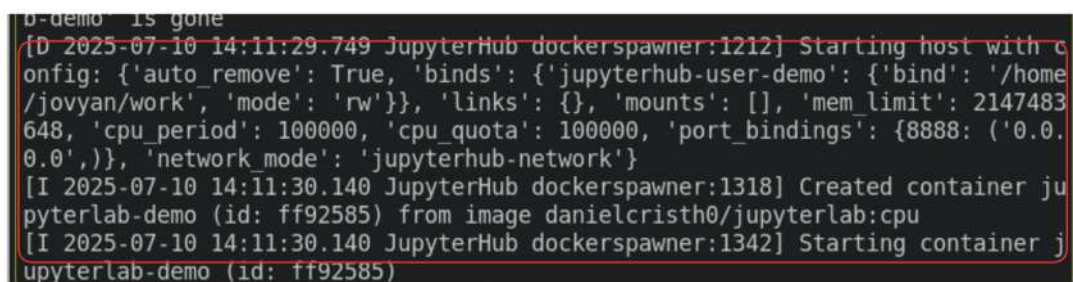
#### 4.1.3.1 Hasil Implementasi MultiNodeSpawner

Hasil implementasi MultiNodeSpawner adalah UI JupyterLab akan diluncurkan di *remote node*. Selanjutnya *node* yang menjalankan JupyterLab UI adalah *node* yang berperan atau memiliki flag "*Primary*", dimana *node* ini adalah *node* yang memiliki *load* (CPU usage dan Memory usage) rendah. Sebagai contoh ketika *user* demo memilih dua *nodes*, maka Jupyterlab UI akan dijalankan di *node rpl*.



Gambar 4.3: Menjalankan JupyterLab UI

Untuk mengetahui apakah JupyterLab UI dijalankan di *node rpl*, bisa melalui log JupyterHub, seperti di Gambar 4.4



Gambar 4.4: Log spawn JupyterLab UI

Terakhir, fungsi dari MultiNodeSpawner adalah *generate file* konfigurasi kernel (kernelspec) untuk setiap *node* yang dipilih pengguna yang nanti akan dijalankan oleh JEG. Contohnya dapat dilihat pada Kode Sumber di bawah ini.

```
1 {
2   "display_name": "Python 3 on rpl",
3   "language": "python",
4   "metadata": {
5     "process_proxy": {
```



```

6         "class_name": "enterprise_gateway.services.processproxies.↵
          distributed.DistributedProcessProxy"
7     },
8     "debugger": true
9 },
10 "argv": [
11     "docker",
12     "run",
13     "--rm",
14     "--network=host",
15     "danielcrith0/jupyterlab:cpu",
16     "python3",
17     "/usr/local/bin/launch_ipykernel.py",
18     "--RemoteProcessProxy.kernel-id",
19     "{kernel_id}",
20     "--RemoteProcessProxy.response-address",
21     "{response_address}",
22     "--RemoteProcessProxy.public-key",
23     "{public_key}",
24     "--RemoteProcessProxy.port-range",
25     "{port_range}",
26     "--RemoteProcessProxy.spark-context-initialization-mode",
27     "none"
28 ],
29 "env": {
30     "NODE_IP": "10.21.73.139",
31 }
32 }

```

---

Kode Sumber 4.3: Contoh KernelSpec

## 4.2 Hasil Uji Coba

### 4.2.1 Menjalankan Single User-Single Node GPU

- Pada Gambar 4.6b, saat pengguna memilih profil "Single GPU", *service discovery* pada JupyterHub akan memfilter dan hanya menampilkan *node* yang memiliki GPU. Hal ini ditunjukkan oleh respons dari *service discovery* yang memastikan bahwa *node* terpilih memiliki atribut `"has_gpu": true`, seperti terlihat pada Kode Sumber 4.2.1.

**Sign Up**

Warning: JupyterHub seems to be served over an unsecured HTTP connection. We strongly recommend enabling HTTPS for JupyterHub.

Username:  
demo

Password:  
demo@123

Confirm password:  
demo@123

**Create User**

[Login](#) with an existing user.

(a) Melakukan Registrasi User

**Sign In**

Warning: JupyterHub seems to be served over an unsecured HTTP connection. We strongly recommend enabling HTTPS for JupyterHub.

Username:  
demo

Password:  
demo@123

**Sign In**

[Sign up](#) to create a new user.

(b) Login dengan User yang telah diregistrasi

Gambar 4.5: Proses Registrasi dan Login pada JupyterHub

## Server Options

✓ Discovery Service Connected

**Select Profile**

**Single Cpu**  
Single node with CPU only  
2 CPU cores 2 GB RAM 1 node

**Single Gpu** (selected)  
Single node with GPU acceleration  
4 CPU cores 8 GB RAM 1 node GPU enabled

**Multi Cpu**  
Multiple nodes with CPU only  
4 CPU cores 8 GB RAM 2-4 nodes

**Multi Gpu**  
Multiple nodes with GPU acceleration  
4 CPU cores 16 GB RAM 2-4 nodes GPU enabled

(a) Memilih profil

## Environment Configuration

Docker Image  
GPU Environment (danielcristh0/jupyterlab:gpu)

Choose CPU for general computing or GPU for machine learning tasks

Node Configuration  
**Single Node**

**Available Nodes**

Node	Status	IP	CPU	Memory	Containers	GPU
rpl (Primary)	Available	10.21.73.139	24 cores	67.11 GB	9 active	Available

CPU USAGE: 1.0% | MEMORY USAGE: 15.6%

**Selection Summary**  
Selected 1 node:  
• Total Resources: 24 CPU cores, 67.11 GB RAM, 1 GPU(s)

**Start**

(b) Memilih environment

Gambar 4.6: User memilih profil dan environment sesuai kebutuhan pada halaman JupyterHub

```

1 {
2   "count": 1,
3   "selected_nodes": [
4     {

```

```

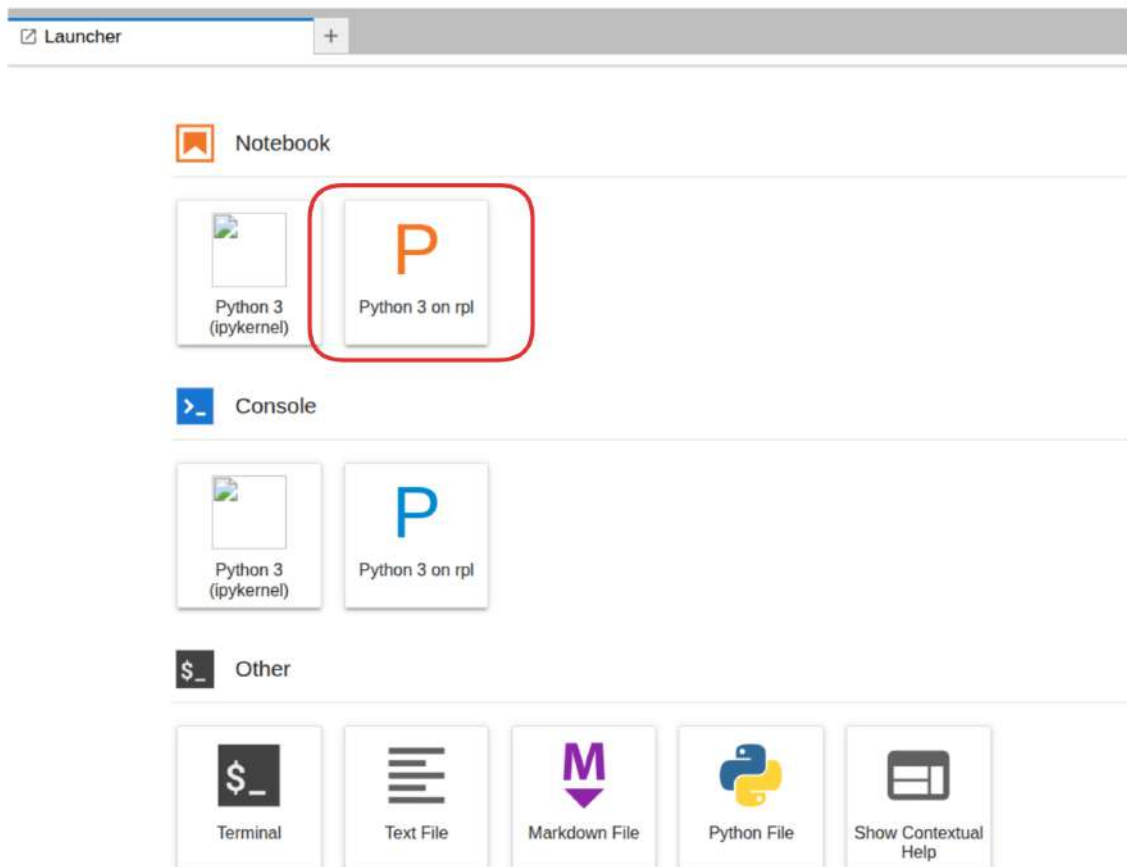
5     "active_jupyterlab": 1,
6     "active_ray": 0,
7     "cpu_cores": 24,
8     "cpu_usage_percent": 1.0,
9     "created_at": "2025-07-08T20:34:14.203389",
10    "disk_usage_percent": 58.5,
11    "gpu_info": [],
12    "has_gpu": true,
13    "hostname": "rpl",
14    "id": 201,
15    "ip": "10.21.73.139",
16    "is_active": true,
17    "load_score": 13.28,
18    "max_containers": 10,
19    "memory_usage_percent": 15.6,
20    "ram_gb": 67.11,
21    "total_containers": 8,
22    "updated_at": "2025-07-10T18:12:15.301315"
23  }
24 ],
25 "status": "ok"
26 }

```

---

Response 4.1: *Response* dari Service Discovery yang hanya menampilkan *node* memiliki GPU sesuai dengan permintaan *user*.

- Ketika user menjalankan kernel "Python 3 on rpl", maka JEG secara otomatis akan meluncurkan *remote kernel* yang berjalan di dalam *container* Docker ke *node* *rpl* yang memiliki alamat IP 10.21.73.139. Hal ini bisa dikonfirmasi dengan melihat *log* pada Gambar 4.8.



Gambar 4.7: Terdapat 1 remote kernel, sesuai dengan jumlah node yang dipilih sebelumnya

```
[I 2025-07-10 10:06:36.789 EnterpriseGatewayApp] Kernel launched on '10.21.73.139', pid: 2605539, ID: d7599724-1377-4a39-af5c-60294d3498a0, Log file: 10.21.73.139:/tmp/kernel-d7599724-1377-4a39-af5c-60294d3498a0.log, Command: ['docker', 'run', '--rm', '--network=host', 'danielcrith0/jupyterlab:cpu', 'python3', '/usr/local/bin/launch_ipykernel.py', '--RemoteProcessProxy.kernel-id', 'd7599724-1377-4a39-af5c-60294d3498a0', '--RemoteProcessProxy.response-address', '10.33.17.30:8877', '--RemoteProcessProxy.public-key', 'MIGfMA0GCsQGSib3DQEBAQUAA4GNADCBiQKBgQD1XL2RgrgzCFFhBQs6MS9QypCMUzhx0NCm6I/G7qn3gM5ZL7z+SVghi7GoYlU0UuvvmHulwKNbnwHpPrNtdVS5XztsS8WtEwFmrdJmkUNHqOS3GGzQZMsxvGtY1kzCKrdkZNY5xjIAteoS/VAVxMRxCHGtjub/mME0AeK75CyS6wIDAQAB', '--RemoteProcessProxy.port-range', '0.0.0.0:0-65535']
```

Gambar 4.8: JEG menjalankan *remote kernel*

- Di Gambar 4.9 setelah *kernel* berhasil dijalankan, pengguna bisa mengakses Jupyter Notebook yang terhubung ke *remote kernel node rpl*. Pengguna bisa menjalankan perintah `!hostname` di *cell* untuk memastikan di *node* mana *kernel*-nya berjalan. Lalu untuk memantau ketersediaan GPU, pengguna bisa menjalankan perintah `!nvidia-smi`.

```
[1]: !hostname
rpl

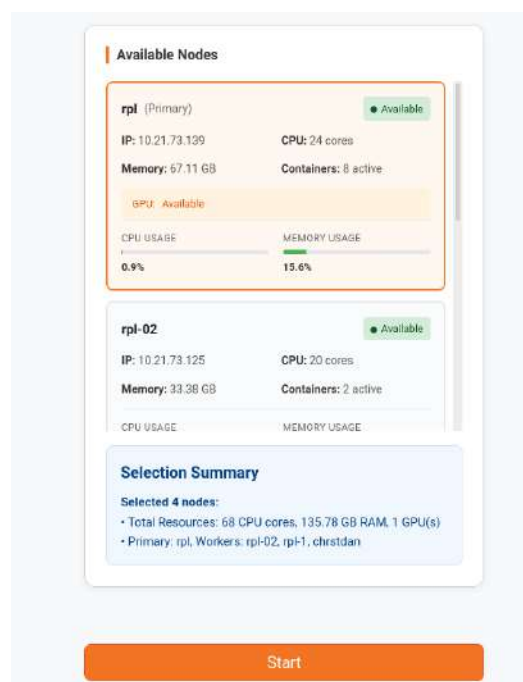
[2]: !nvidia-smi
Thu Jul 10 09:52:18 2025
+-----+
| NVIDIA-SMI 570.144                Driver Version: 570.144      CUDA Version: 12.8     |
+-----+-----+-----+-----+-----+-----+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|-----+-----+-----+-----+-----+-----+
| 0   NVIDIA GeForce RTX 3080 Ti    Off      | 00000000:01:00:00 Off |         0%      Default |
| 30%   31C    P8              30W / 350W | 10532MiB / 12288MiB |           |         N/A |
+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:                       |
| GPU  GI  CI           PID  Type  Process name                        | GPU Memory |
|  ID   ID                 |          |          |                      | Usage       |
+-----+-----+-----+-----+-----+-----+
|                                     |                                     |                                     |                                     |
+-----+-----+-----+-----+-----+-----+
+-----+

```

Gambar 4.9: Interaksi yang dilakukan di Jupyter Kernel

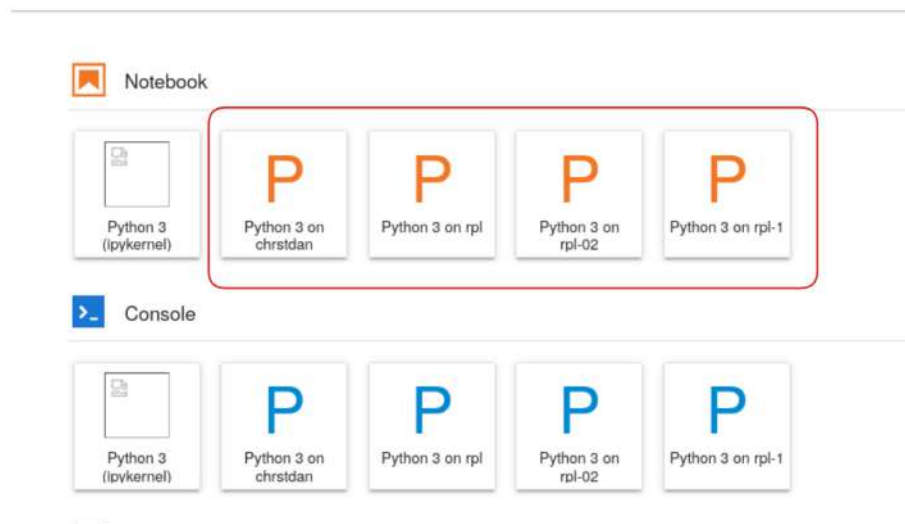
## 4.2.2 Menjalankan Single User-Multi Nodes CPU

Pada pengujian ini jumlah node yang dipilih adalah 4 (jumlah maksimal), pemilihan *nodes* dapat dilihat di gambar di bawah ini



Gambar 4.10: Spawn Multi Nodes CPU

Ketika di-*start* maka hasilnya di JupyterLab UI akan ada 4 *kernels* dengan nama *kernel* menggunakan *hostname* dari tiap *node*



Gambar 4.11: Multi Nodes Kernel

### 4.2.3 Uji Coba Multi-User Concurrent

Pengujian ini bertujuan untuk memvalidasi kemampuan sistem dalam menangani permintaan dari banyak pengguna secara bersamaan. Percobaan ini akan dilakukan dalam beberapa simulasi dengan menggunakan **Locust** untuk mensimulasikan multi-user concurrent.

**Locust** digunakan sebagai alat uji beban dengan mengatur jumlah pengguna *virtual* dan tingkat permintaan per detik (*request per second*). Setiap pengguna *virtual* melakukan proses *login* ke JupyterHub, memilih profil, dan melakukan *spawn* JupyterLab. Parameter yang divariasikan meliputi jumlah pengguna simultan dan waktu ramp-up. Hasil pengujian mencakup waktu respon, tingkat keberhasilan spawn, dan beban pada masing-masing *node*.

Pengujian dilakukan dengan memanfaatkan infrastruktur yang telah dirinci pada Tabel 3.7, di mana perangkat laptop berperan sebagai *node manager* yang menjalankan komponen JupyterHub dan Discovery Service. Perangkat ini juga mengatur distribusi permintaan pengguna ke beberapa node yang tersedia melalui mekanisme *service discovery*. Lalu Komputer RPL, RPL 1 dan RPL 2 akan digunakan sebagai tempat mendistribusikan *container* JupyterLab.

#### 4.2.3.1 Simulasi Menggunakan 10 User selama 10 Detik

Simulasi pertama dilakukan dengan melibatkan 10 pengguna *virtual* yang diatur menggunakan **Locust**. Setiap pengguna melakukan proses *login*, memilih profil komputasi, dan melakukan *spawn* JupyterLab melalui JupyterHub.

Jumlah pengguna dan kecepatan *ramp-up* diatur sebesar 1 pengguna per detik, dengan durasi pengujian selama 10 detik dengan alamat **<http://10.33.17.30:18000>** yang merupakan alamat dari JupyterHub. Pengujian ini ditujukan untuk mengamati perilaku dasar sistem saat menerima permintaan serentak dari beberapa pengguna, tanpa memaksakan beban tinggi. Berdasarkan hasil awal, tidak ditemukan adanya kegagalan (*failure*) pada proses permintaan, dan seluruh pengguna berhasil melewati proses login dan spawn dengan sukses selama 10 detik.

LOCUST

Host: http://10.33.17.30:18000 | Status: READY | RPS: 0 | Failures: 0%

### Start new load test

Number of users (peak concurrency) \*  
1

Ramp up (users started/second) \*  
1

Host  
http://10.33.17.30:18000

Advanced options

Run time (e.g. 20s, 20m, 3m, 2h, 1h20m, 3h30m10s, etc.)  
10

Profile

START

Gambar 4.12: Opsi untuk memilih load simulasi menggunakan 10 *user*

LOCUST

Host: http://10.33.17.30:18000 | Status: RUNNING | Users: 10 | RPS: 2.9 | Failures: 69%

EDIT STOP RESET

STATISTICS CHARTS FAILURES EXCEPTIONS CURRENT RATIO DOWNLOAD DATA LOGS LOCUST CLOUD

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
POST	/hub/login	10	10	320	340	340	319.64	308	337	6737.7	1	1
GET	/hub/login [GET]	10	0	10	22	22	10.98	7	22	7948	1	0
POST	/hub/spawn	12	12	1000	1100	1100	878.46	18	1064	8999.58	0.9	0.9
Aggregated		32	22	320	1100	1100	432.74	7	1064	7964.13	2.9	1.9

Gambar 4.13: Proses *load test user*

Proses membuat *user* lalu meluncurkan *container* Jupyterlab adalah 10 detik dimulai dari pukul 04:23:21 sampai selesai di pukul 04:23:33.

```
[2025-07-11 04:22:48,461] chrstdan/INFO/locust.main: Starting web interface at http://0.0.0.0:8089, press enter to open your default browser.
[2025-07-11 04:23:21,568] chrstdan/INFO/locust.runners: Ramping to 10 users at a rate of 1.00 per second
[testuser6] Memulai virtual user...
[testuser6] Mulai proses pemilihan profil...

[2025-07-11 04:23:30,578] chrstdan/INFO/locust.runners: All users spawned: {"JupyterHubUser": 10} (10 total users)
[testuser10] Memulai virtual user...
```

Gambar 4.14: Durasi *load test* untuk menjalankan 10 *user*

Total ada 10 *user* yang di *create* selama 10 detik simulasi, sesuai dengan jumlah *user* yang dibuat pada konfigurasi Locust.

▼ testuser1		13 seconds ago	Stop Server	Access Server	Edit User
▼ testuser2		12 seconds ago	Stop Server	Access Server	Edit User
▼ testuser3		7 seconds ago	Stop Server	Access Server	Edit User
▼ testuser4		9 seconds ago	Stop Server	Access Server	Edit User
▼ testuser7		14 seconds ago	Stop Server	Access Server	Edit User
▼ testuser6		15 seconds ago	Stop Server	Access Server	Edit User
▼ testuser5		8 seconds ago	Stop Server	Access Server	Edit User
▼ testuser9		11 seconds ago	Stop Server	Access Server	Edit User
▼ testuser10		6 seconds ago	Stop Server	Access Server	Edit User
▼ testuser8		10 seconds ago	Stop Server	Access Server	Edit User

Gambar 4.15: *User* yang terdaftar di halaman admin JupyterHub

#### 4.2.3.2 Simulasi Menggunakan 20 *User* selama 10 Detik

Simulasi dilakukan dengan menambahkan 10 *user* baru, lalu untuk pilihan *run time* tetap 10 detik. Simulasi ini akan menjalankan 20 *user*, lalu memilih profil "Multi Node CPU" dan meluncurkan *container* JupyterLab selama 10 detik. Proses *load test* berjalan pada pukul 05:16:27 dan selesai di pukul 05:20:32.

Untuk distribusi *container* JupyterLab ternyata hanya dijalankan di 1 *node*, yaitu *node* rpl. Dimana total dari 29 *container* yang aktif 20 diantaranya adalah *container* JupyterLab. Hal ini tidak sesuai dengan yang diharapkan, yang mana harusnya *container* JupyterLab bisa didistribusikan ke *node* lain yang memiliki jumlah *container* kurang dari 20.

```

Zellij (sparkling-brachiosaur) - danielpepuho@chrstdan:/mnt/nvme0n1p11/Gith...
Zellij (sparkling-brachiosaur) Tab #1 Tab #2 +2 →
ssh daniel@10.21.73.139
daniel@rpl:~$ docker ps | wc -l
29
daniel@rpl:~$

```

Gambar 4.16: Total *container* yang aktif di *node* rpl



*[Halaman ini sengaja dikosongkan]*

## BAB V

### PENUTUP

#### 5.1 Kesimpulan

Berdasarkan hasil pengujian yang telah dilakukan pada BAB IV, diperoleh kesimpulan sebagai berikut:

1. Sistem berhasil dirancang untuk mengintegrasikan JupyterHub sebagai antarmuka dengan *service discovery*, sehingga setiap *instance* JupyterLab dapat dijalankan di *node* yang memiliki sumber daya paling sesuai. Pemilihan *node* dilakukan secara dinamis berdasarkan informasi *load* dari setiap *node* yang dilaporkan secara berkala oleh Agent Service.
2. Pengujian awal dengan 10 pengguna secara bersamaan menunjukkan bahwa sistem mampu menangani permintaan dan mendistribusikan *container* tanpa kendala. Namun, pengujian terhadap skenario dengan beban lebih tinggi masih perlu dilakukan untuk menjawab rumusan masalah kedua secara menyeluruh.

#### 5.2 Saran

Untuk pengembangan lebih lanjut pada sistem integrasi JupyterHub, Service Discovery, dan Jupyter Enterprise Gateway antara lain:

1. Pengujian dengan jumlah pengguna lebih besar dan variasi beban perlu dilakukan untuk mengevaluasi skalabilitas dan ketahanan sistem secara menyeluruh untuk menjawab rumusan masalah kedua.
2. Mengembangkan fitur *discovery* histori pemilihan *node* sebagai dasar untuk evaluasi dan penyempurnaan mekanisme penjadwalan.
3. Perlu dilakukan perbaikan pada integrasi antara halaman JupyterHub dan Jupyter Enterprise Gateway agar pemilihan *node* benar-benar selaras dengan penempatan *kernel*. Saat ini, *kernel* masih berjalan pada remote host yang tidak selalu sesuai dengan *node* yang dipilih pengguna.
4. Mekanisme komunikasi antara Spawner, JupyterHub, dan JEG perlu ditinjau ulang agar informasi *node* yang dipilih dapat diteruskan secara konsisten hingga ke proses *kernel launching*.
5. Halaman JupyterHub dapat dikembangkan lebih lanjut agar lebih informatif dan interaktif, misalnya dengan menampilkan status *node* secara *real-time*, antrian pengguna, dan estimasi waktu spawn. Hal ini akan membantu pengguna dalam memilih *node* yang sesuai dengan kebutuhannya.

*[Halaman ini sengaja dikosongkan]*

## DAFTAR PUSTAKA

- Docker Documentation. (2025). *Docker documentation* [Accessed: 16 July 2025]. <https://docs.docker.com/compose>
- Flask Pallets Team. (2025). *Flask documentation* [Accessed: 20 June 2025]. <https://flask.palletsprojects.com/en/stable>
- Heroku Team. (2025). *Websocket security* [Accessed: 15 July 2025]. <https://devcenter.heroku.com/articles/websocket-security>
- JEG Development Team. (2025). *Jupyter enterprise documentation* [Accessed: 20 June 2025]. <https://jupyter-enterprise-gateway.readthedocs.io/en/latest/>
- Jupyter Team. (2025). *Jeg documentation* [Accessed: 20 June 2025]. [https://github.com/jupyter-server/enterprise\\_gateway/tree/main](https://github.com/jupyter-server/enterprise_gateway/tree/main)
- Kumar, A., Cuccuru, G., Grüning, B., & Backofen, R. (2023). An accessible infrastructure for artificial intelligence using a docker-based jupyterlab in galaxy [Published: 26 April 2023]. *GigaScience*, 12. <https://doi.org/10.1093/gigascience/giad028>
- PostgreSQL Development Team. (2025). *Postgresql documentation* [Accessed: 20 June 2025]. <https://www.postgresql.org/docs/>
- Redis Team. (2025). *Redis documentation* [Accessed: 20 June 2025]. <https://redis.io/docs/latest/>
- Shikai Wang, X. W., Haotian Zheng, & Shang, F. (2024). Distributed high-performance computing methods for accelerating deep learning training. *jklst*. <https://jklst.org/index.php/home/article/view/230>
- Team, J. (2024). *Jupyterhub: Technical overview* [Accessed: May 30 2025]. <https://jupyterhub.readthedocs.io/en/latest/reference/technical-overview.html>
- Team, J. D. (2024). *Jupyterlab documentation* [Accessed: December 13, 2024]. <https://jupyterlab.readthedocs.io/en/latest>
- Team, S. D. (2024). *What is docker architecture?* [Accessed: December 24, 2024]. <https://sysdig.com/learn-cloud-native/what-is-docker-architecture>
- Turnbull, J. (2014). *The docker book: Containerization is the new virtualization*.
- Zhou, N., Zhou, H., & Hoppe, D. (2022). Containerisation for high performance computing systems: Survey and prospects. *arXiv*. <https://arxiv.org/abs/2212.08717>
- ZMQ Authors. (2025). *Zmq documentation* [Accessed: 15 July 2025]. <https://zeromq.org/get-started>

*[Halaman ini sengaja dikosongkan]*

## BIOGRAFI PENULIS



Gloriyano Cristho Daniel Pepuho, lahir di Nabire pada 19 Agustus 2002. Penulis menempuh pendidikan formal di SD YPK Sion Nabire, SMP YPPK St. Antonius Nabire, dan SMKN 1 Sentani. Pada tahun 2020, penulis diterima sebagai mahasiswa di Departemen Teknik Informatika, FTEIC-ITS.

Selama menempuh studi, penulis aktif dalam berbagai kegiatan akademik dan pengembangan proyek. Penulis menjadi Administrator Laboratorium Networking Technology and Intelligent Cybersecurity (NETICS) dari tahun 2022 hingga 2024, di mana penulis turut membantu kegiatan pembelajaran sebagai asisten dosen pada mata kuliah Sistem Operasi dan Jaringan Komputer. Selain itu, penulis juga berpartisipasi sebagai staf Divisi IT dalam kegiatan Schematics serta terlibat dalam proyek pengembangan sistem Penerimaan Peserta Didik Baru (PPDB) Online untuk SMA/SMK se-Jawa Timur selama periode 2022–2025.

*[Halaman ini sengaja dikosongkan]*