

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 Department of Informatics Engineering

Department of Department of Informatics Engineering

Faculty of Intelligent Electrical and Informatics Technology

Sepuluh Nopember Institute of Technology

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)

.....

Dr. Galileo Galilei, S.T., M.Sc.
NIP: 18560710 194301 1 001

(Penguji I)

.....

Friedrich Nietzsche, S.T., M.Sc.
NIP: 18560710 194301 1 001

(Penguji II)

.....

SURABAYA
Juni, 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 Department of Informatics Engineering
Department of Department of Informatics Engineering
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)

.....

Dr. Galileo Galilei, S.T., M.Sc.
NIP: 18560710 194301 1 001

(Examiner I)

.....

Friedrich Nietzsche, S.T., M.Sc.
NIP: 18560710 194301 1 001

(Examiner II)

.....

SURABAYA
June, 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, June 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 : Department of Informatics Engineering
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, June 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.

Dalam era teknologi yang semakin maju, kebutuhan akan komputasi berbasis GPU menjadi sangat penting, khususnya dalam bidang kecerdasan buatan (AI) dan analisis data skala besar. GPU memungkinkan pemrosesan paralel yang cepat dan efisien, sehingga sering digunakan untuk melatih model deep learning dan menjalankan tugas-tugas komputasi intensif. Namun, pengelolaan GPU di lingkungan multi-pengguna menghadapi tantangan besar, seperti alokasi sumber daya yang tidak merata dan potensi penurunan efisiensi sistem. Untuk mengatasi masalah ini, penelitian ini bertujuan untuk mengembangkan mekanisme penjadwalan GPU yang efisien dengan memanfaatkan teknologi Docker Container dan antarmuka JupyterLab. Docker digunakan untuk menciptakan lingkungan kerja yang terisolasi bagi setiap pengguna, sementara JupyterLab menyediakan platform interaktif yang memudahkan pengguna dalam mengakses dan menjalankan tugas berbasis GPU secara simultan. Penelitian ini dibagi kedalam beberapa tahap yang meliputi analisis kebutuhan, desain sistem, serta perancangan metode evaluasi. Rancangan sistem yang diusulkan akan diimplementasikan pada kluster GPU di lingkungan laboratorium atau institusi pendidikan. Evaluasi direncanakan mencakup pengujian efisiensi alokasi sumber daya, kemudahan akses pengguna, dan skalabilitas sistem dalam mendukung banyak pengguna secara bersamaan. Penelitian ini diharapkan dapat memberikan kontribusi terhadap pengelolaan sumber daya GPU dalam lingkungan komputasi terdistribusi, mendukung efisiensi dan keadilan alokasi, serta meningkatkan pengalaman pengguna dalam mengakses sumber daya GPU untuk kebutuhan komputasi modern.

Kata Kunci: *Kluster 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.

In the era of advancing technology, the demand for GPU-based computing has become increasingly critical, particularly in the fields of artificial intelligence (AI) and large-scale data analysis. GPUs enable fast and efficient parallel processing, making them widely used for training deep learning models and performing computationally intensive tasks. However, managing GPUs in multi-user environments presents significant challenges, such as uneven resource allocation and potential system inefficiencies. To address these issues, this study aims to develop an efficient GPU scheduling mechanism utilizing Docker container technology and the JupyterLab interface. Docker creates isolated work environments for each user, while JupyterLab provides an interactive platform that simplifies simultaneous GPU-based task execution. The research consists of several phases, including requirement analysis, system design, and evaluation method planning. The proposed system design will be implemented on a GPU cluster in a laboratory or educational institution environment. Evaluation will include testing resource allocation efficiency, user accessibility, and system scalability in supporting multiple concurrent users. This study is expected to make a significant contribution to GPU resource management in distributed computing environments, promoting efficiency and fairness in resource allocation while enhancing the user experience in accessing GPU resources for modern computational needs.

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, Juni 2025

Gloriyano Cristho Daniel Pepuho

[Halaman ini sengaja dikosongkan]

DAFTAR ISI

ABSTRAK	i
ABSTRACT	iii
KATA PENGANTAR	v
DAFTAR ISI	vii
DAFTAR GAMBAR	ix
DAFTAR TABEL	xi
DAFTAR KODE SUMBER	xiii
DAFTAR SINGKATAN	xv
1 PENDAHULUAN	1
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	3
1.6 Sistematika Penulisan	3
2 TINJAUAN PUSTAKA	5
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
2.1.3 Syndeo: Portable Ray Clusters with Secure Containerization	6
2.1.4 Ray: A Distributed Framework for Emerging AI Applications	6
2.2 Teori/Konsep Dasar	7
2.2.1 Klaster GPU	7

2.2.2	Docker	7
2.2.3	JupyterLab	9
2.2.4	JupyterHub	9
2.2.5	Jupyter Enterprise Gateway	12
2.2.6	Ray	12
2.2.7	Flask	14
2.2.8	Redis	14
2.2.9	PostgreSQL	14
3	DESAIN DAN IMPLEMENTASI	15
3.1	Perancangan Arsitektur Sistem	16
3.1.1	Service Discovery	18
3.1.2	Service Agent	18
3.1.3	JupyterHub	18
3.1.4	Ray Cluster	18
3.2	Implementasi Sistem	19
3.2.1	Service Discovery	19
3.2.2	Service Agent	31
3.2.3	JupyterHub	35
3.2.4	Ray Cluster	39
3.3	Peralatan Pendukung	40
4	PENGUJIAN DAN ANALISIS	43
4.1	Hasil dan Pengujian	43
4.1.1	Uji Akses dan Proses Spawn Server	43
4.1.2	Skenario 2: Multi-User Concurrent	46
5	PENUTUP	47
5.1	Kesimpulan	47
5.2	Saran	47
	DAFTAR PUSTAKA	49
	BIOGRAFI PENULIS	51

DAFTAR GAMBAR

2.1	Arsitektur Docker (Sumber: S. D. Team, 2024)	8
2.2	Arsitektur <i>JupyterHub</i> (Sumber: J. D. Team, 2024)	10
2.3	Komponen <i>RAY</i> (Sumber: Moritz et al., 2018)	13
3.1	Arsitektur Penelitian	17
3.2	Hasil build agent menjadi image	33
3.3	Log dari container agent ketika melakukan registrasi node	34
4.1	Proses Registrasi dan Login pada JupyterHub	44
4.2	User memilih profil dan environment sesuai kebutuhan pada halaman konfigurasi JupyterHub	44
4.3	Tahapan proses spawning container setelah konfigurasi dilakukan oleh user	45
4.4	JupyterLab berhasil dijalankan di lingkungan multi-node hingga mencapai tampilan akhir yang siap digunakan	45

[Halaman ini sengaja dikosongkan]

DAFTAR TABEL

3.1	Struktur Direktori Discovery Service	19
3.2	Kolom dan Tipe Data Model Node	22
3.3	Kolom dan Tipe Data Model NodeSelection	22
3.4	Kolom dan Tipe Data Model NodeMetric	23
3.5	Contoh Isi File .env dari <i>Discovery Service</i>	27
3.6	Daftar Endpoint REST API pada Discovery Service	30
3.7	Struktur Direktori Proyek JupyterHub	35
3.8	Spesifikasi Peralatan Pendukung	40
3.9	Daftar Perangkat Lunak Pendukung	41

[Halaman ini sengaja dikosongkan]

DAFTAR KODE SUMBER

3.1	Pseudocode untuk <code>app.py</code>	20
3.2	Pseudocode untuk Model Node	21
3.3	Pseudocode untuk Model NodeSelection	22
3.4	Pseudocode untuk Model NodeMetric	22
3.5	Pseudocode untuk Koneksi Redis Menggunakan ConnectionPool	23
3.6	Pseudocode Penyimpanan Data Node ke Redis dengan TTL	24
3.7	Pseudocode Fungsi Perhitungan Skor Node	25
3.8	Pseudocode untuk <code>config.py</code>	25
3.9	Dockerfile Service Discovery	28
3.10	Docker Compose Service Discovery	28
3.11	Menjalankan Discovery Service via Docker Compose	30
3.12	Fungsi Register Agent	31
3.13	Kumpulan Informasi Sistem oleh Agent	32
3.14	Deteksi Container JupyterLab dan Ray	32
3.15	Deteksi GPU Menggunakan <code>gpustat</code>	32
3.16	Loop Registrasi Agent Tiap 15 Detik	32
3.17	Dockerfile untuk Agent Service	33
3.18	Perintah untuk Build dan Menjalankan Agent	33
3.19	Perintah untuk Build dan Menjalankan Agent	33
3.20	Perintah untuk Menjalankan Agent di setiap Node	33
3.21	<code>jupyterhub_config.py</code>	36
3.22	Pemanggilan API Seleksi Node	36
3.23	Inisialisasi Docker Client Berdasarkan IP Node	37
3.24	Pembuatan Container Ray Worker di Node Tambahan	37
3.25	Eksekusi Paralel untuk Container Jupyter dan Worker	37
3.26	Override URL pada PatchedMultiNodeSpawner	38

[Halaman ini sengaja dikosongkan]

DAFTAR SINGKATAN

API	Application Programming Interface
CPU	Central Processing Unit
CLI	Command Line Interface
GPU	Graphics Processing Unit
GUI	Graphical User Interface
JEG	Jupyter Enterprise Gateway
PAAS	Platform as a Service

[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 eksponensial. Urgensi pergeseran ini ditegaskan oleh CEO NVIDIA, Jensen Huang, dalam pidato utamanya 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 berkat kemampuannya dalam pemrosesan paralel yang cepat dan efisien 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 dalam keadaan *idle*. Untuk memaksimalkan potensi sumber daya yang ada, muncul kebutuhan untuk memanfaatkan infrastruktur GPU yang tersebar ini secara kolektif.

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.

Dalam tugas akhir ini, dikembangkan sebuah sistem untuk mengelola penggunaan infrastruktur GPU yang tersebar bagi banyak pengguna. Sistem ini memanfaatkan teknologi containerisasi menggunakan Docker untuk menciptakan lingkungan kerja yang terisolasi dan konsisten. Sebagai antarmuka utama, JupyterLab digunakan untuk menyediakan platform yang interaktif dan mudah diakses, memungkinkan pengguna menjalankan tugas komputasi tanpa perlu konfigurasi yang rumit.

Dengan demikian, penelitian ini berfokus pada pengembangan mekanisme penjadwalan GPU yang efektif dalam lingkungan terdistribusi. Sistem yang diusulkan dirancang untuk dapat mengelola dan mendistribusikan beban kerja secara dinamis ke node-node yang tersedia, termasuk yang sebelumnya dalam keadaan idle, guna mendukung kebutuhan komputasi AI modern yang semakin kompleks dan kolaboratif.

1.2 Rumusan Masalah

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

1. Bagaimana cara mengimplementasikan mekanisme penjadwalan sumber daya (resource scheduling) yang efektif pada lingkungan komputasi dinamis, di mana ketersediaan dan beban setiap node GPU dapat berubah sewaktu-waktu?
2. Bagaimana arsitektur integrasi antara JupyterHub dan service discovery dirancang untuk memungkinkan setiap instance JupyterLab dapat memanfaatkan sumber daya komputasi (GPU dan CPU) yang tersebar di berbagai node secara efisien?
3. Bagaimana sistem menangani dan mendistribusikan antrian permintaan dari banyak pengguna yang masuk secara bersamaan (concurrent requests), dan bagaimana pengaruhnya terhadap alokasi sumber daya pada setiap node?
4. Bagaimana kinerja sistem dalam menyelesaikan sejumlah tugas komputasi secara simultan, dan bagaimana mekanisme penanganan kegagalan (failure handling) diterapkan untuk memastikan reliabilitas saat terjadi masalah pada salah satu node atau kontainer?

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 platform cloud komersial 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 orchestrator lain seperti 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.
4. Penanganan kegagalan (failure handling) terbatas pada deteksi node atau kontainer yang tidak responsif dan tidak mencakup mekanisme pemulihan state atau checkpointing tugas yang kompleks.

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 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 kluster GPU, teknologi Docker, Ray Framework, penjadwalan GPU, JupyterLab, dan JupyterHub. Bab ini menjadi landasan teoritis untuk melakukan pengembangan sistem.

3. BAB III Desain dan Implementasi Sistem

Bab ini berisi perancangan arsitektur sistem yang mencakup service discovery, integrasi JupyterHub, dan konfigurasi Ray cluster. 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 perform 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 desain antarmuka berbasis *container* dapat meningkatkan efisiensi dan aksesibilitas sistem (Kumar et al., 2023).

2.1.3 Syndeo: Portable Ray Clusters with Secure Containerization

Pada paper ini, peneliti memperkenalkan *Syndeo*, sebuah framework untuk mengelola dan mengorkestrasi cluster RAY secara portable menggunakan *container*. Fokus utama dari penelitian ini adalah bagaimana *Syndeo* dapat memanfaatkan kontainerisasi untuk meningkatkan portabilitas, keamanan, dan skalabilitas dalam menjalankan *workload* RAY di berbagai platform cloud, seperti AWS, Azure, dan Google Cloud. Framework ini dirancang untuk mendukung komputasi *throughput* tinggi *multi-node* dan memastikan keamanan dengan membatasi hak istimewa pengguna, sehingga administrator memiliki kontrol penuh atas akses sistem. *Syndeo* juga memungkinkan implementasi *workflow paralell* Ray pada sistem manajemen kluster seperti *Slurm*, yang sebelumnya tidak didukung secara *native*.

Temuan dalam paper ini relevan dengan penelitian ini, terutama dalam konteks penggunaan Docker dan Ray untuk mengelola sumber daya GPU dalam kluster terdistribusi. *Syndeo* memberikan wawasan tentang pentingnya portabilitas, keamanan, dan orkestrasi yang efisien dalam lingkungan multi-pengguna, yang dapat menjadi inspirasi untuk pengelolaan pengguna dan alokasi sumber daya dalam sistem berbasis Kluster yang digunakan pada penelitian ini (Li et al., 2024).

2.1.4 Ray: A Distributed Framework for Emerging AI Applications

Dalam paper ini, peneliti memperkenalkan Ray, sebuah framework terdistribusi yang dirancang untuk mendukung aplikasi AI modern, seperti *reinforcement learning* dan *deep learning*. Framework ini menawarkan antarmuka terpadu yang mampu mengekspresikan komputasi berbasis tugas (*task-parallel*) dan aktor (*actor-based*), didukung oleh mesin eksekusi dinamis tunggal. Ray mengimplementasikan penjadwalan terdistribusi dan penyimpanan yang toleran terhadap kesalahan untuk mengelola status kontrol sistem. Eksperimen yang dilakukan menunjukkan bahwa Ray dapat menskalakan hingga lebih dari 1,8 juta tugas per detik dan memberikan kinerja yang lebih baik dibandingkan sistem khusus lainnya.

Temuan dari paper ini relevan dengan penelitian ini dalam beberapa aspek. Pertama, Ray sebagai framework terdistribusi untuk aplikasi AI sesuai dengan kebutuhan penelitian untuk memanfaatkan teknologi tersebut dalam pengelolaan infrastruktur GPU terdistribusi. Kedua, kemampuan Ray dalam mendukung model komputasi *task-parallel* dan *actor-based* memberikan fleksibilitas yang diperlukan dalam penjadwalan dan alokasi sumber daya GPU di lingkungan multi-pengguna. Ketiga, fitur penjadwalan terdistribusi dan toleransi kesalahan pada Ray dapat meningkatkan efisiensi dan keandalan sistem yang dikembangkan. Keempat, skalabilitas tinggi Ray, yang mampu menangani jutaan tugas per detik, relevan untuk mendukung penggunaan GPU oleh banyak pengguna secara simultan (Moritz et al., 2018).

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

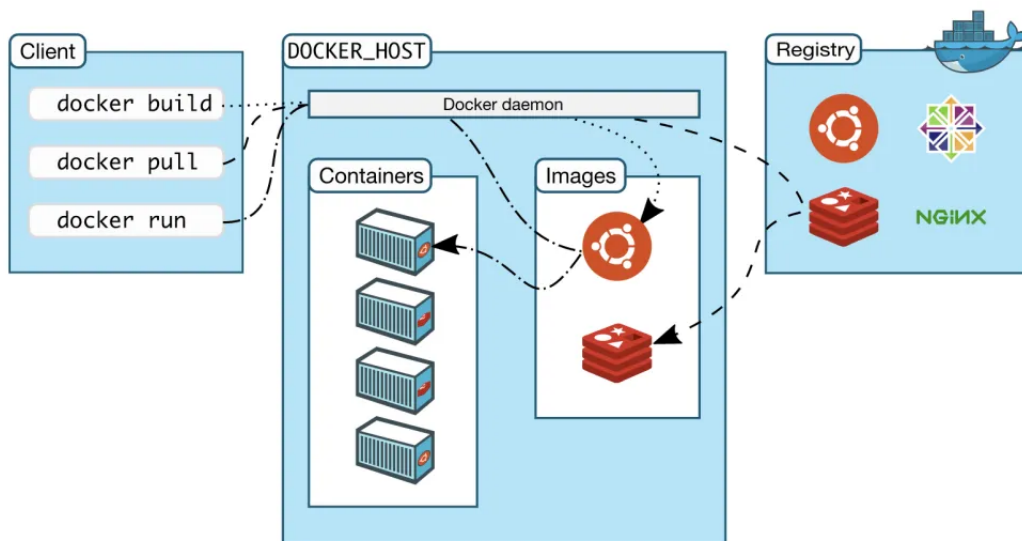
Container. Image ini mencakup semua dependensi, pustaka, dan file yang diperlukan untuk menjalankan aplikasi dalam container. Pengguna dapat membuat *images* dari *Dockerfile* atau *pull images* yang sudah ada dari Docker Hub atau *registry* lainnya. *Images* bersifat modular dan dapat *diupdate* atau 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)

2.2.3 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.

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 Ray framework memungkinkan pengguna menjalankan komputasi terdistribusi langsung dari notebook tanpa konfigurasi manual yang kompleks.

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.

A. Komponen Utama JupyterLab:

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. **Extension System:** Akses terminal penuh yang terintegrasi dalam antarmuka web, memungkinkan eksekusi perintah sistem langsung dari browser.
4. **Terminal:** Mendukung plugin dan ekstensi untuk memperluas fungsionalitas sesuai kebutuhan spesifik pengguna.

2.2.4 JupyterHub

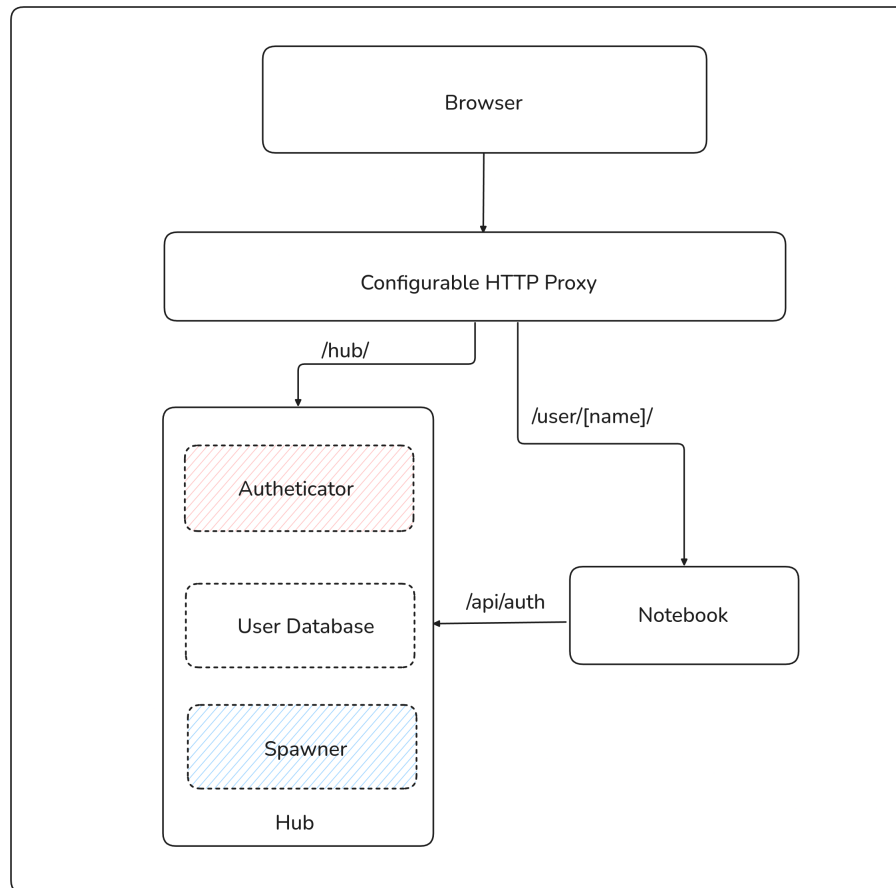
JupyterHub adalah platform open-source yang memungkinkan banyak pengguna untuk mengakses dan menjalankan lingkungan Jupyter Notebook secara terisolasi melalui antarmuka web. Dirancang untuk mendukung skenario multi-pengguna, JupyterHub sangat cocok digunakan dalam lingkungan pendidikan, penelitian, dan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

A. Arsitektur Utama JupyterHub

JupyterHub terdiri dari tiga komponen utama yang bekerja secara sinergis:

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. D. Team, 2024)

B. Alur Kerja JupyterHub

Proses interaksi pengguna dengan JupyterHub dapat dijelaskan sebagai berikut:

1. **Akses Awal:** Pengguna mengakses JupyterHub melalui browser web dengan menggunakan alamat IP atau nama domain yang telah dikonfigurasi.
2. **Autentikasi:** Data login yang dimasukkan oleh pengguna dikirim ke komponen **Autheticator** untuk validasi. Jika valid, pengguna akan dikenali dan diizinkan untuk melanjutkan.
3. **Peluncuran Server Notebook:** Setelah autentikasi berhasil, JupyterHub akan meluncurkan instance server notebook khusus 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 dengan lingkungan Jupyter seperti biasa.

C. 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:** Menggunakan Pluggable Authentication Modules (PAM) dari sistem operasi host.
 - **OAuthAuthenticator:** Mendukung autentikasi menggunakan OAuth2, seperti Github, Google, atau GitLab.
 - **LDAPAuthenticator:** Terintegrasi dengan sistem direktori LDAP untuk autentikasi berbasis domain.
 - **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 tergantung pada sistem eksternal.
- **Spawner:** Mengontrol cara peluncuran server notebook untuk setiap pengguna. Beberapa jenis Spawner yang umum digunakan antara lain:
 - **BatchSpawner:** Menyediakan integrasi dengan sistem manajemen antrian pekerjaan seperti SLURM atau PBS. Spawner ini cocok untuk lingkungan komputasi dengan resource terbatas dan kebutuhan scheduling yang ketat.
 - **DockerSpawner:** Menjalankan server notebook dalam container Docker, memberikan isolasi lingkungan yang lebih baik.
 - **KubeSpawner:** Menggunakan Kubernetes untuk mengelola dan menskalakan server notebook di lingkungan kluster.
 - **MultiNodeSpawner (kustom 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.

2.2.5 Jupyter Enterprise Gateway

Jupyter Enterprise Gateway (JEG) adalah komponen open-source dari ekosistem Project Jupyter 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, YARN, atau Hadoop.

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 worker node yang memiliki resource sesuai kebutuhan pengguna. Hal ini meningkatkan skalabilitas dan memungkinkan penggunaan sumber daya yang lebih optimal dalam arsitektur komputasi terdistribusi.

A. Fitur Utama JEG

1. Remote Kernel Execution: Memungkinkan kernel notebook berjalan di node remote tanpa perlu instalasi kernel di node frontend.
2. Multi-tenancy: Mendukung banyak pengguna untuk mengeksekusi kernel di environment yang terisolasi.
3. Scalability: Dapat diintegrasikan dengan cluster manager (misal Kubernetes atau 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.

2.2.6 Ray

Ray adalah framework open-source yang dirancang untuk membangun dan menjalankan aplikasi komputasi paralel serta terdistribusi secara efisien.

Ray menyediakan abstraksi tingkat tinggi yang memungkinkan pengembang mengeksekusi tugas paralel melalui dua paradigma pemrograman utama, yaitu:

1. **Task-based Computing (Stateless):** Memungkinkan fungsi-fungsi Python dijalankan secara paralel menggunakan `@ray.remote`. Paradigma ini cocok untuk proses yang dapat dibagi menjadi unit-unit kecil independen.
2. **Actor-based Computing (Stateful):** Digunakan untuk komputasi yang membutuhkan state yang dipertahankan selama proses berjalan. Cocok untuk layanan yang berjalan terus-menerus atau berbasis shared-state.

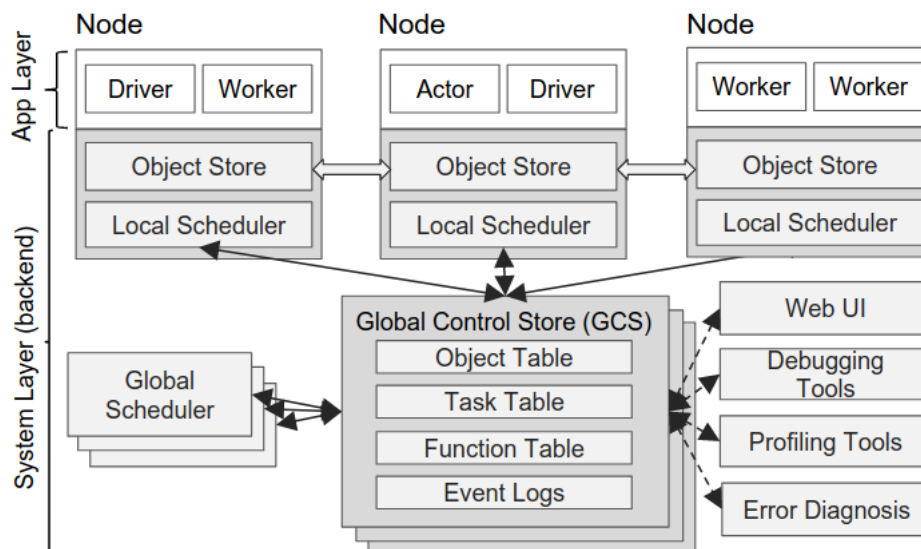
Ray mampu mengeksekusi jutaan tugas per detik berkat desain sistemnya yang efisien dan toleran terhadap kesalahan. Eksperimen dalam paper Moritz et al. (2018) menunjukkan bahwa Ray dapat menskalakan lebih dari 1.8 juta tugas per detik dengan latensi rendah.

A. Arsitektur Ray

Ray terdiri dari dua lapisan utama, yaitu Application Layer dan System Layer:

- **Application Layer** mencakup komponen:
 - *Driver*: Mengatur alur eksekusi program.
 - *Worker*: Menjalankan fungsi stateless secara paralel.
 - *Actor*: Menjalankan komponen stateful.
- **System Layer (Backend)**:
 - *Object Store*: Penyimpanan memori efisien dan zero-copy.
 - *Raylet*: Scheduler lokal dan pengelola transfer objek.
 - *Global Control Store (GCS)*: Menyimpan metadata dan koordinasi antar-node.
 - *Global Scheduler*: Menentukan penempatan tugas lintas node.

Ray cocok digunakan dalam sistem ini karena mampu menjalankan berbagai workload AI seperti pelatihan model deep learning, reinforcement learning, hingga pemrosesan data paralel di lingkungan multi-pengguna. Selain itu, Ray juga mudah diintegrasikan dengan JupyterLab, sehingga memungkinkan pengguna mengakses komputasi terdistribusi secara interaktif langsung dari notebook.



Gambar 2.3: Komponen RAY (Sumber: Moritz et al., 2018)

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.

Pada sistem ini, Flask digunakan untuk membangun Discovery Service, yaitu layanan yang menerima dan menyediakan informasi status real-time dari seluruh node dalam kluster GPU. 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 blueprint.

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

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*.

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, kepatuhan terhadap standar SQL, serta dukungan fitur tingkat lanjut seperti transaksi ACID, indexing kompleks, dan extensibility.

Pada proyek ini, PostgreSQL digunakan sebagai penyimpanan persisten untuk metadata 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.

BAB III

DESAIN DAN IMPLEMENTASI

Bab ini menjelaskan perancangan dan implementasi sistem pengelolaan sumber daya GPU secara terdistribusi menggunakan container Docker, JupyterHub, dan Ray. 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 Perancangan Arsitektur Sistem

Penelitian ini diawali dengan proses perancangan sistem yang bertujuan untuk memungkinkan pengelolaan sumber daya 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.

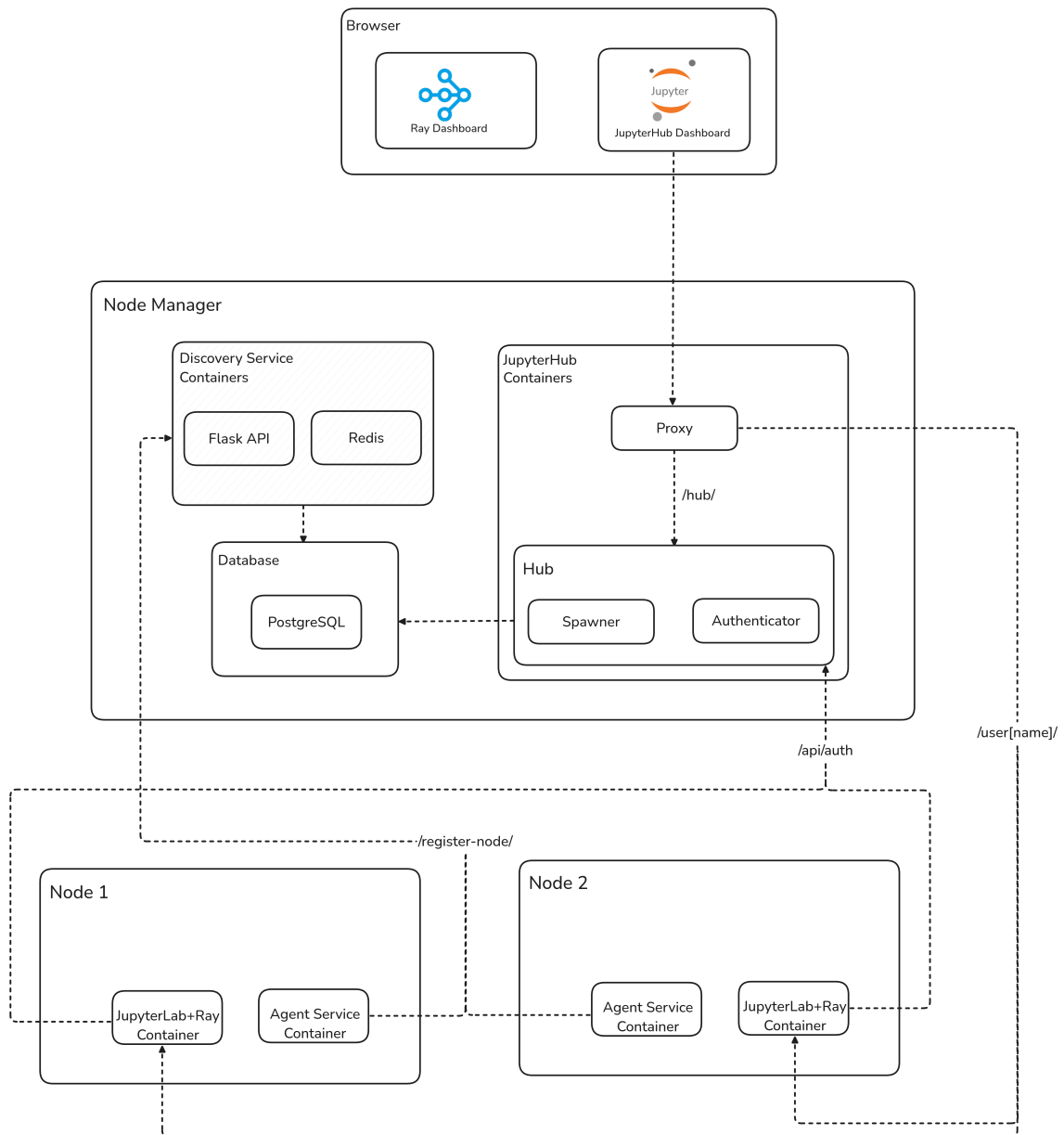
Langkah awal dalam perancangan adalah merancang sistem service discovery yang berfungsi sebagai pusat pengumpulan dan penyimpanan status sumber daya dari seluruh node yang tersedia dalam klaster. Data ini mencakup informasi penggunaan CPU, RAM, keberadaan GPU, serta jumlah container aktif, yang dikirim 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 penjadwalan 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. Pemilihan node dilakukan secara dinamis dan adaptif, bergantung pada profil sumber daya yang dibutuhkan oleh masing-masing pengguna.

Dalam *environment* kontainer yang di-spawn, pengguna akan mendapatkan akses ke antarmuka JupyterLab secara personal dan terisolasi. Di dalam container tersebut, Ray Worker dijalankan secara otomatis dan terhubung ke Ray Head Node. Dengan demikian, setiap pengguna dapat langsung menjalankan komputasi paralel menggunakan framework Ray tanpa memerlukan konfigurasi manual tambahan.

Untuk mengimplementasikan arsitektur tersebut, dilakukan konfigurasi Discovery Service, Agent Service, JupyterHub, dan Ray secara terintegrasi. Masing-masing komponen dikemas dalam Docker container untuk memudahkan deployment dan orkestrasi antar layanan. Komunikasi antar komponen dilakukan melalui protokol HTTP REST API dan jaringan internal antar container.

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 dan orkestrasi komputasi dengan Ray.



Gambar 3.1: Arsitektur Penelitian

Adapun komponen utama yang membentuk arsitektur sistem ini terdiri atas:

3.1.1 Service Discovery

Service discovery bertugas sebagai pusat informasi status terkini dari setiap node dalam kluster. Informasi yang dikumpulkan meliputi status CPU, RAM, GPU, dan jumlah container aktif. Data ini dikumpulkan secara periodik oleh agen yang berjalan di setiap node dan dikirim ke Redis melalui REST API Flask. Redis berfungsi sebagai basis data cepat (in-memory) untuk mendukung pengambilan keputusan real-time saat pemilihan node terbaik untuk menjalankan JupyterLab user.

3.1.2 Service Agent

Service Agent merupakan komponen ringan 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 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.

3.1.4 Ray Cluster

Ray digunakan untuk mengatur workload komputasi paralel. Dalam perancangannya, setiap container JupyterLab pengguna akan menjalankan Ray Worker secara otomatis dan terhubung ke Ray Head Node. Dengan cara ini, pengguna dapat langsung menggunakan fitur komputasi terdistribusi seperti `ray.remote()` tanpa konfigurasi manual. Ray menjembatani antar-node agar task berat bisa dijalankan dengan efisien di GPU atau CPU sesuai kapasitas.

3.2 Implementasi Sistem

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

3.2.1 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.

A. 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 kustom.
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.

B. 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: Pseudocode untuk *app.py*

C. 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 **Flask-Migrate**.

```
1 CLASS Node:
2     ATTRIBUTE id : Integer // primary key
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 = True
10    ATTRIBUTE max_containers : Integer = 10
11    ATTRIBUTE created_at : Datetime = Now
12    ATTRIBUTE updated_at : Datetime = Now (on update)
13    ATTRIBUTE last_heartbeat : Datetime
14 END CLASS
```

Kode Sumber 3.2: Pseudocode untuk Model Node

Tabel 3.2: Kolom dan Tipe Data Model Node

Nama Kolom	Tipe Data
id	Integer (pk)
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
last_heartbeat	Datetime

```

1 CLASS NodeSelection:
2     ATTRIBUTE id : Integer // primary key
3     ATTRIBUTE profile_id : Integer // foreign key to Profile
4     ATTRIBUTE user_id : String(255) // indexed
5     ATTRIBUTE session_id : String(255)
6     ATTRIBUTE selected_nodes : JSON
7     ATTRIBUTE selection_reason : String(50)
8     ATTRIBUTE created_at : Datetime = Now // indexed
9 END CLASS

```

Kode Sumber 3.3: Pseudocode untuk Model NodeSelection

Tabel 3.3: Kolom dan Tipe Data Model NodeSelection

Nama Kolom	Tipe Data
id	Integer (pk)
profile_id	Integer (fk)
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 // primary key
3     ATTRIBUTE node_id : Integer // foreign key to Node
4     ATTRIBUTE cpu_usage_percent : Float

```

```

5     ATTRIBUTE memory_usage_percent : Float
6     ATTRIBUTE disk_usage_percent : Float
7     ATTRIBUTE active_jupyterlab : Integer = 0
8     ATTRIBUTE active_ray : Integer = 0
9     ATTRIBUTE total_containers : Integer = 0
10    ATTRIBUTE load_score : Float
11    ATTRIBUTE recorded_at : Datetime = Now // indexed
12 END CLASS

```

Kode Sumber 3.4: Pseudocode untuk Model NodeMetric

Tabel 3.4: Kolom dan Tipe Data Model NodeMetric

Nama Kolom	Tipe Data
id	Integer (pk)
node_id	Integer (fk)
cpu_usage_percent	Float
memory_usage_percent	Float
disk_usage_percent	Float
active_jupyterlab	Integer (default: 0)
active_ray	Integer (default: 0)
total_containers	Integer (default: 0)
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.
- Menentukan apakah node masih aktif berdasarkan heartbeat agent.

Koneksi ke Redis dilakukan melalui file `redis_client.py` menggunakan *connection pool* untuk efisiensi koneksi. File ini diakses melalui kelas `RedisService` pada `services/redis_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: Pseudocode 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: Pseudocode Penyimpanan Data Node ke Redis dengan TTL

D. 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 `calculate_node_score` 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: Pseudocode 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.

E. 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") == "↵
    true"
18
19  // Redis Configuration
20  ATTRIBUTE REDIS_HOST = getenv("REDIS_HOST", "localhost")
21  ATTRIBUTE REDIS_PORT = getenv("REDIS_PORT", 6379) AS Integer
22  ATTRIBUTE REDIS_PASSWORD = getenv("REDIS_PASSWORD", "redis@pass")
23  ATTRIBUTE REDIS_EXPIRE_SECONDS = getenv("REDIS_EXPIRE_SECONDS", 45) ↵
    AS Integer
24
25  // Load Balancer Thresholds
26  ATTRIBUTE DEFAULT_MAX_CPU_USAGE = 80.0
27  ATTRIBUTE DEFAULT_MAX_MEMORY_USAGE = 85.0
28  ATTRIBUTE STRICT_MAX_CPU_USAGE = 60.0
29  ATTRIBUTE STRICT_MAX_MEMORY_USAGE = 60.0
30  ATTRIBUTE STRICT_MAX_CONTAINERS = 5
31
32  // Scoring Weights & Penalties
33  ATTRIBUTE CPU_WEIGHT = 0.8
34  ATTRIBUTE MEMORY_WEIGHT = 0.8
35  ATTRIBUTE HEAVY_PENALTY = 80
36  ATTRIBUTE MEDIUM_PENALTY = 20
37  END CLASS

```

Kode Sumber 3.8: Pseudocode 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.

F. 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. Berkas Dockerfile berikut akan membangun image Python 3.12, menginstal dependensi dari `requirements.txt`, dan menjalankan `app.py` sebagai aplikasi utama.

```
1 # syntax=docker/dockerfile:1.3
2
3 FROM python:3.12-slim
4
5 ENV PYTHONDONTWRITEBYTECODE=1 \
6     PYTHONUNBUFFERED=1 \
7     TZ=Asia/Jakarta
8
9 WORKDIR /app
10
11 COPY requirements.txt .
12 RUN pip install --no-cache-dir -r requirements.txt
13
14 COPY . .
15
16 EXPOSE 15002
17 CMD ["python", "app.py"]
```

Kode Sumber 3.9: Dockerfile Service Discovery

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

```
1 version: "3.8"
2
3 services:
4   discovery:
5     build:
6       context: .
7       dockerfile: Dockerfile
8     container_name: discovery-api
9     restart: unless-stopped
10    ports:
11      - "15002:15002"
12    environment:
13      POSTGRES_HOST: postgres
14      POSTGRES_PORT: 5432
15      POSTGRES_DB: 'voyager'
16      POSTGRES_USER: postgres
17      POSTGRES_PASSWORD: postgres
18
19      REDIS_HOST: redis
20      REDIS_PORT: 16379
21      REDIS_PASSWORD: "redis@pass"
22      REDIS_EXPIRE_SECONDS: 45
23
24      API_HOST: 0.0.0.0
```

```

25     API_PORT: 15002
26     DEBUG: "True"
27     SECRET_KEY: "secret-service111111"
28     depends_on:
29       - postgres
30       - redis
31     networks:
32       - discovery-network
33
34     postgres:
35       image: postgres:14-alpine
36       container_name: postgres
37       restart: unless-stopped
38       ports:
39         - "5432:5432"
40       environment:
41         POSTGRES_DB: voyager
42         POSTGRES_USER: postgres
43         POSTGRES_PASSWORD: postgres
44         POSTGRES_INITDB_ARGS: "--encoding=UTF-8"
45         TZ: Asia/Jakarta
46       volumes:
47         - postgres_data:/var/lib/postgresql/data
48         - ./init.sql:/docker-entrypoint-initdb.d/init.sql
49       healthcheck:
50         test: ["CMD-SHELL", "pg_isready -U postgres -d voyager"]
51         interval: 10s
52         timeout: 5s
53         retries: 5
54         start_period: 30s
55       networks:
56         - discovery-network
57
58     redis:
59       image: redis:7-alpine
60       container_name: redis
61       restart: unless-stopped
62       volumes:
63         - redis_data:/data
64         - ./redis.conf:/usr/local/etc/redis/redis.conf
65       command: ["redis-server", "/usr/local/etc/redis/redis.conf"]
66       environment:
67         TZ: Asia/Jakarta
68       ports:
69         - "16379:16379"
70       healthcheck:
71         test: ["CMD", "redis-cli", "-p", "16379", "ping"]
72         interval: 10s
73         timeout: 3s
74         retries: 3
75       networks:
76         - discovery-network
77
78     volumes:
79       postgres_data:
80       redis_data:
81

```

```

82 networks:
83     discovery-network:
84         driver: bridge

```

Kode Sumber 3.10: Docker Compose Service Discovery

`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

G. List API Endpoint

Tabel 3.6: 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.

3.2.2 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.

A. Arsitektur dan Fungsi Agent

Agent dikembangkan sebagai skrip Python mandiri yang berjalan sebagai *container* pada setiap node. Agent ini dirancang agar:

- Mengirimkan data sistem setiap 15 detik.
- Menangkap informasi hardware dan aktivitas container.
- Tetap ringan dan tidak membebani node secara signifikan.

B. Implementasi dan Pengumpulan Data

Agent diimplementasikan dalam bahasa Python dan berjalan sebagai *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 Listing 3.12. Fungsi ini bertugas mengumpulkan data menggunakan `collect_node_info()` dan mengirimkannya ke API.

```
1 def register():
2     payload = collect_node_info()
3     if payload:
4         resp = requests.post(DISCOVERY_URL, json=payload)
```

Kode Sumber 3.12: 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 (JupyterLab dan Ray).
- Deteksi GPU NVIDIA.

```

1 def collect_node_info():
2     hostname = socket.gethostname()
3     ip_address = os.popen("hostname -I").read().strip().split()[0]
4     ram_gb = round(psutil.virtual_memory().total / 1e9, 2)
5     cpu_usage = psutil.cpu_percent(interval=1)
6     memory = psutil.virtual_memory()
7     disk = psutil.disk_usage("/")

```

Kode Sumber 3.13: Kumpulan Informasi Sistem oleh Agent

Agent juga menghitung jumlah container yang berjalan dengan membaca nama dan image-nya. Hal ini dilakukan oleh fungsi `get_container_info()`, yang akan mengenali apakah container tersebut merupakan JupyterLab atau Ray Worker.

```

1 def get_container_info():
2     containers = docker_client.containers.list()
3     for container in containers:
4         if "jupyter" in container.name or "jupyter" in container.image.tags:
5             ...

```

Kode Sumber 3.14: Deteksi Container JupyterLab dan Ray

Untuk mendeteksi keberadaan GPU, agent menggunakan pustaka `gpustat`. Jika GPU NVIDIA tersedia, maka informasi seperti penggunaan memori, suhu, dan load GPU akan dikirimkan. Jika tidak tersedia, akan dilakukan fallback untuk deteksi AMD GPU.

```

1 def get_gpu_stats():
2     stats = gpustat.GPUStatCollection.new_query()
3     for gpu in stats.gpus:
4         gpu_info.append({
5             "name": gpu.name,
6             "memory_used_mb": gpu.memory_used,
7             "utilization_gpu_percent": gpu.utilization
8         })

```

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 __name__ == "__main__":
2     while True:
3         register()
4         time.sleep(15)

```

Kode Sumber 3.16: Loop Registrasi Agent Tiap 15 Detik

C. 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 3.17 menunjukkan isi file Dockerfile yang digunakan untuk membangun image Agent.

```
1 FROM python:3.12-slim
2
3 ENV PYTHONDONTWRITEBYTECODE=1 \
4     PYTHONUNBUFFERED=1 \
5     TZ=Asia/Jakarta
6
7 WORKDIR /app
8
9 COPY requirements.txt .
10 RUN pip install --no-cache-dir -r requirements.txt
11
12 COPY . .
13
14 EXPOSE 15002
15 CMD ["python", "agent.py"]
```

Kode Sumber 3.17: Dockerfile untuk Agent Service

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

Selanjutnya agent akan di-build menjadi Docker image dan akan di-push ke Docker registry:

```
1 # Build image agent
2 docker build -t danielcristh0/agent:1.1 .
```

Kode Sumber 3.18: Perintah untuk Build dan Menjalankan Agent



Image Name	Tag	ID	Time Since Build	Size
danielcristh0/agent	1.1	0c646f042897	25 hours ago	134MB

Gambar 3.2: Hasil build agent menjadi image

```
1
2 # Push image agent
3 docker push danielcristh0/agent:1.1
```

Kode Sumber 3.19: Perintah untuk Build dan Menjalankan Agent

```
1
2 # Menjalankan agent di setiap node
3 docker run --name agent -d \
4     --net=host \
5     -e DISCOVERY_URL=http://192.168.122.1:15002/register-node \
6     danielcristh0/agent:1.1
```

```

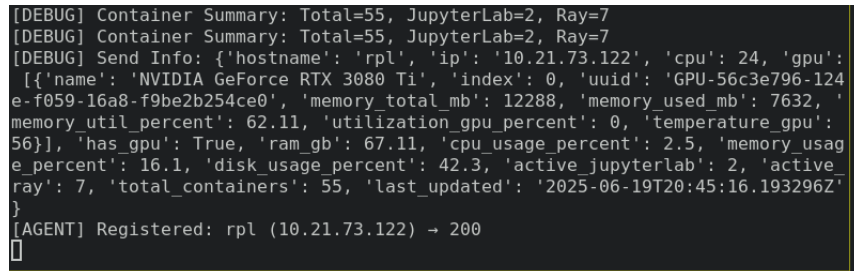
7
8 # Tambahkan "--gpus all" di node yang memiliki GPU
9 docker run --name agent -d \
10     --net=host \
11     -e DISCOVERY_URL=http://10.21.73.116:15002/register-node \
12     -v /var/run/docker.sock:/var/run/docker.sock \
13     --gpus all \
14     danielcristh0/agent:1.1

```

Kode Sumber 3.20: Perintah untuk Menjalankan Agent di setiap Node

Agent akan mengirimkan informasi ke endpoint **/register-node** dan menggunakan IP Address node manager. Penggunaan mode **-net=host** memungkinkan agent mengakses informasi IP node dengan benar serta membaca container aktif melalui Docker daemon lokal.

Docker logs dari container agent:



```

[DEBUG] Container Summary: Total=55, JupyterLab=2, Ray=7
[DEBUG] Container Summary: Total=55, JupyterLab=2, Ray=7
[DEBUG] Send Info: {'hostname': 'rpl', 'ip': '10.21.73.122', 'cpu': 24, 'gpu':
[{'name': 'NVIDIA GeForce RTX 3080 Ti', 'index': 0, 'uuid': 'GPU-56c3e796-124
e-f059-16a8-f9be2b254ce0', 'memory_total_mb': 12288, 'memory_used_mb': 7632, '
memory_util_percent': 62.11, 'utilization_gpu_percent': 0, 'temperature_gpu':
56}], 'has_gpu': True, 'ram_gb': 67.11, 'cpu_usage_percent': 2.5, 'memory_usag
e_percent': 16.1, 'disk_usage_percent': 42.3, 'active_jupyterlab': 2, 'active
ray': 7, 'total_containers': 55, 'last_updated': '2025-06-19T20:45:16.193296Z'
}
[AGENT] Registered: rpl (10.21.73.122) → 200

```

Gambar 3.3: Log dari container agent ketika melakukan registrasi node

3.2.3 JupyterHub

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.

Pada bagian ini akan dijelaskan struktur proyek, konfigurasi sistem, serta integrasi multi-node spawner dengan Service Discovery.

A. 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 3.7: 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.

B. 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 from config.env import load_environment
2 from config.hub import configure_hub
3 from config.spawner import configure_spawner
4 from config.proxy import configure_proxy
5 from config.auth import configure_auth
6 from config.hooks import attach_hooks
7
8 load_environment(c)
9 configure_hub(c)
10 configure_spawner(c)
11 configure_proxy(c)
12 configure_auth(c)
13 attach_hooks()
```

Kode Sumber 3.21: `jupyterhub_config.py`

3.2.3.1 Integrasi Multi-Node dan Spawner Khusus

Untuk mendukung eksekusi JupyterLab pada beberapa node sekaligus, sistem ini mengimplementasikan spawner kustom berbasis `DockerSpawner`, yang disebut `MultiNodeSpawner`. Seluruh kode terkait ditempatkan dalam direktori **`spawner/`**.

A. 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`, dan menggunakan node pertama sebagai tempat menjalankan JupyterLab utama, sementara node lainnya digunakan untuk Ray Worker.

```
1 payload = {
2     "profile_id": self.profile_id,
3     "user_id": self.user.name,
4     "num_nodes": self.num_nodes
5 }
6 response = requests.post(f"{self.discovery_api_url}/select-nodes", json=payload)
7 self.selected_nodes = response.json().get("selected_nodes", [])
```

Kode Sumber 3.22: Pemanggilan API Seleksi Node

Setiap node yang dipilih akan dihubungi melalui koneksi Docker remote. Untuk itu, spawner membuat klien Docker dinamis berdasarkan ip node dengan format `tcp://<ip>:2375`. Fungsi

ini di-override melalui metode `_docker()`.

```
1 def _docker(self, node_ip=None):
2     if not node_ip:
3         return super()._docker()
4     if node_ip not in self._docker_clients:
5         self._docker_clients[node_ip] = docker.DockerClient(base_url=f"↵
        tcp://{node_ip}:2375")
6     return self._docker_clients[node_ip]
```

Kode Sumber 3.23: Inisialisasi Docker Client Berdasarkan IP Node

Setelah node terpilih, proses pembuatan container dibagi dua:

1. **Container JupyterLab primary**: dijalankan di node pertama, berisi environment inter-aktif untuk pengguna.
2. **Container JupyterLab worker(opsional)**: dijalankan di node-node sisanya jika profil pengguna mendukung komputasi paralel.

Fungsi `create_worker_container()` akan mengatur nama, image, environment, dan volume binding untuk container Ray Worker. Informasi ID container disimpan di `worker_containers[us` untuk keperluan manajemen.

```
1 worker_container = docker_client.containers.run(
2     image=image,
3     name=container_name,
4     command="ray start --address={ip}:{port}",
5     environment=worker_env,
6     detach=True
7 )
```

Kode Sumber 3.24: Pembuatan Container Ray Worker di Node Tambahan

Semua proses `create_user_container()` dan `create_worker_container()` dijalankan secara paralel dengan `asyncio.gather()` agar proses spawning lebih cepat dan efisien.

```
1 await asyncio.gather(
2     self.create_user_container(primary_node, image),
3     *(self.create_worker_container(n, image) for n in self.selected_nodes↵
        [1:])
4 )
```

Kode Sumber 3.25: Eksekusi Paralel untuk Container Jupyter dan Worker

B.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 3.26 menunjukkan implementasi override URL pada PatchedMultiNodeSpawner:

```
1 @property
2 def url(self):
3     base_url = self.server_url
4     if hasattr(self, 'default_url') and self.default_url:
5         base_url += self.default_url.lstrip("/")
6     return base_url
7
8 @property
9 def server_url(self):
10    if self.ip and self.port:
11        return f"http://{self.ip}:{self.port}"
12    return ""
```

Kode Sumber 3.26: Override URL pada PatchedMultiNodeSpawner

3.2.4 Ray Cluster

Ray digunakan untuk mengatur workload komputasi paralel. Dalam perancangannya, setiap container JupyterLab pengguna akan menjalankan Ray Worker secara otomatis dan terhubung ke Ray Head Node. Dengan cara ini, pengguna dapat langsung menggunakan fitur komputasi terdistribusi seperti `ray.remote()` tanpa konfigurasi manual. Ray menjembatani antar-node agar task berat bisa dijalankan dengan efisien di GPU atau CPU sesuai kapasitas.

Bagian ini akan menjelaskan cara integrasi Ray ke dalam sistem, termasuk konfigurasi head dan worker node, serta bagaimana komputasi paralel dapat dijalankan langsung dari dalam JupyterLab.

3.3 Peralatan Pendukung

Perangkat yang digunakan untuk pengerjaan tugas akhir ini merupakan sebuah komputer dengan spesifikasi sebagai berikut.

Tabel 3.8: Spesifikasi Peralatan Pendukung

No.	Komponen	Spesifikasi
1	Laptop	
	<i>Brand</i>	Asus
	<i>Processor</i>	AMD Ryzen 3
	<i>Operating System</i>	Ubuntu 22.04 LTS
	<i>GPU</i>	AMD Radeon vega 3 graphics
	<i>Memory</i>	18 GB
	<i>Storage</i>	512 GB
2	Komputer	
	<i>Brand</i>	Asus
	<i>Processor</i>	12th Gen Intel i9-12900K
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>GPU</i>	NVIDIA GeForce RTX 3080 Ti
	<i>Memory</i>	64 GB
	<i>Storage</i>	723 GB
3	Virtual Machine 1	
	<i>Brand</i>	Asus
	<i>Processor</i>	12th Gen Intel i9-12900K
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>GPU</i>	NVIDIA GeForce RTX 3080 Ti
	<i>Memory</i>	64 GB
	<i>Storage</i>	723 GB
4	Virtual Machine 2	
	<i>Brand</i>	Asus
	<i>Processor</i>	12th Gen Intel i9-12900K
	<i>Operating System</i>	Ubuntu 24.04 LTS
	<i>GPU</i>	NVIDIA GeForce RTX 3080 Ti
	<i>Memory</i>	64 GB
	<i>Storage</i>	723 GB

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

Tabel 3.9: 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.
JupyterHub	5.3.0	Menangani autentikasi pengguna serta spawn container JupyterLab ke node terpilih berdasarkan data dari service discovery.
Ray	2.46	Framework komputasi paralel dan terdistribusi. Setiap pengguna dapat langsung menjalankan task terdistribusi secara otomatis dari dalam JupyterLab.
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 service discovery berupa REST API yang menerima dan menyediakan data status node.
Python	3.11	Bahasa pemrograman utama yang digunakan untuk seluruh komponen sistem, seperti konfigurasi JupyterHub, pengembangan REST API, Ray, serta skrip monitoring.
PostgreSQL	14	Basis data relasional yang digunakan untuk menyimpan data historis seperti riwayat pemilihan node, profil pengguna, dan metrik performa dari setiap node.

[Halaman ini sengaja dikosongkan]

BAB IV

PENGUJIAN DAN ANALISIS

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 dan Pengujian

4.1.1 Uji Akses dan Proses Spawn Server

Langkah pertama yang diuji adalah memastikan sistem dapat diakses melalui antarmuka JupyterHub oleh pengguna biasa. Setelah pengguna berhasil login, sistem akan menampilkan form pemilihan profil dan node. Form ini mengirimkan data ke spawner untuk memulai proses alokasi sumber daya dan peluncuran container.

A. Langkah Pengujian

1. Akses halaman JupyterHub melalui `http://<ip_jupyterhub>:18000`.
2. Registrasi dan login menggunakan akun pengguna (misal: `demo`).
3. Pilih profil komputasi dan jumlah node pada form yang tersedia, lalu klik tombol `Launch Server`.

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

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 4.1: 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
Single node with GPU acceleration
2 CPU cores 2 GB RAM 1 node GPU enabled

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

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

(a) Memilih profil

Environment Configuration

Docker Image
CPU Environment (danielcrish0/jupyterlab.cpu)

Choose CPU for general computing or GPU for machine learning tasks

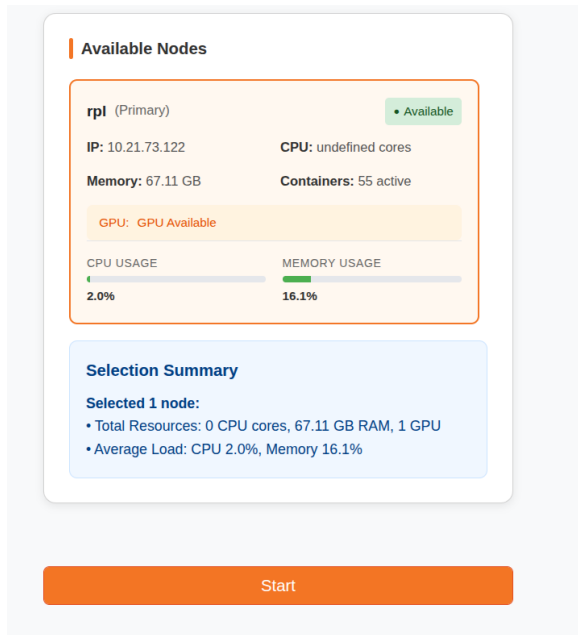
Node Configuration
Single Node Multi Node

3 Nodes - Medium cluster

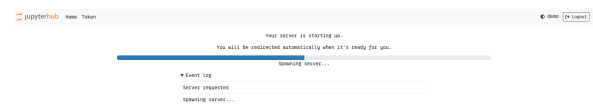
Additional nodes will be used for distributed computing

(b) Memilih environment

Gambar 4.2: User memilih profil dan environment sesuai kebutuhan pada halaman konfigurasi JupyterHub

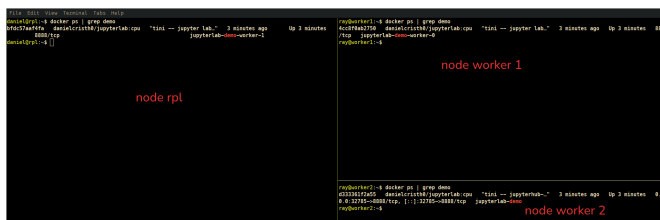


(a) User melakukan spawn container

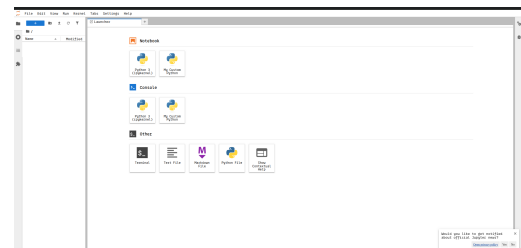


(b) Proseses spawning container

Gambar 4.3: Tahapan proses spawning container setelah konfigurasi dilakukan oleh user



(a) JupyterLab dijalankan di 3 node



(b) Tampilan JupyterLab setelah proses spawn selesai

Gambar 4.4: JupyterLab berhasil dijalankan di lingkungan multi-node hingga mencapai tampilan akhir yang siap digunakan

Hasil yang Diharapkan

Setelah tombol ditekan, sistem akan:

- Memanggil Discovery API untuk memilih node terbaik.
- Menjalankan container JupyterLab di node terpilih.
- Jika profil mendukung Ray, container Ray Worker juga dijalankan paralel.
- Pengguna diarahkan ke halaman JupyterLab dengan environment aktif.

Hasil Aktual

Hasil pengujian menunjukkan bahwa seluruh proses berjalan sesuai harapan. Gambar 4.1b

menunjukkan tampilan halaman awal untuk registrasi dan login, sementara Gambar ?? menunjukkan tampilan JupyterLab yang berhasil dijalankan.

4.1.2 Skenario 2: Multi-User Concurrent

- **Tujuan:** Menguji distribusi kontainer saat 5 user masuk secara paralel.
- **Profil:** 2 user GPU, 3 user CPU.
- **Hasil:** Node GPU digunakan optimal, node CPU terdistribusi merata.

BAB V

PENUTUP

5.1 Kesimpulan

Berdasarkan hasil pengujian yang Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. sebagai berikut:

1. Mekanisme penjadwalan sumber daya dinamis berhasil diterapkan melalui arsitektur Service Discovery dan Service Agent. Service Agent yang berjalan di setiap node secara periodik mengumpulkan dan melaporkan metrik vital—seperti utilisasi CPU, RAM, GPU, dan jumlah kontainer aktif—ke Service Discovery. Data ini memungkinkan sistem untuk menghitung skor beban dan memilih node dengan beban paling optimal untuk setiap permintaan pengguna baru
2. Arsitektur sistem yang mengintegrasikan JupyterHub dengan Service Discovery melalui MultiNodeSpawner kustom telah terbukti fungsional. Spawner ini mampu memanggil API untuk mendapatkan rekomendasi node terbaik, sehingga proses alokasi sumber daya terjadi secara otomatis dan dinamis berdasarkan profil yang dipilih pengguna. Hasil pengujian menunjukkan sistem berhasil menangani permintaan pengguna secara konkuren, mendistribusikan beban kerja secara merata, dan menjalankan kontainer JupyterLab pada node-node yang telah ditentukan
3. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna.

5.2 Saran

Untuk pengembangan lebih lanjut pada Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. antara lain:

1. Memperbaiki Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus.
2. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa.
3. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna.

[Halaman ini sengaja dikosongkan]

DAFTAR PUSTAKA

- 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>
- Li, W., Lafuente Mercado, R. S., Pena, J. D., & Allen, R. E. (2024). Syndeo: Portable ray clusters with secure containerization [arXiv:2409.17070v1 [cs.DC]]. *arXiv preprint arXiv:2409.17070*. <https://arxiv.org/abs/2409.17070>
- Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & Stoica, I. (2018). Ray: A distributed framework for emerging ai applications. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, 561–577. <https://www.usenix.org/conference/osdi18/presentation/moritz>
- 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. D. (2024). *Jupyterhub: Technical overview* [Accessed: May 30 2025]. <https://jupyterhub.readthedocs.io/en/latest/reference/technical-overview.html>
- 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>

[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 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]