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

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) .....
Alan Turing, ST., MT. NIP: 18560710 194301 1 001	(Penguji III) .....

Mengetahui,
Kepala Departemen Departemen Teknik Informatika FTEIC - ITS

Prof. Albus Percival Wulfric Brian Dumbledore, S.T., M.T.
NIP. 18810313 196901 1 001

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) .....
Alan Turing, ST., MT. NIP: 18560710 194301 1 001	(Examiner III) .....

Acknowledged,
Head of Department of Informatics Engineering Department FTEIC - ITS

Prof. Albus Percival Wulfric Brian Dumbledore, S.T., M.T.
NIP. 18810313 196901 1 001

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. NIP. 19810620 200501 1 003	Gloriyano Cristho Daniel Pepuho 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 klaster 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: ***Klaster GPU, Docker Container, JupyterLab, Pengelolaan pengguna***

[Halaman ini sengaja dikosongkan]

ABSTRACT

Name : Gloriyano Cristho Daniel Pepuho
Title : *Managing Distributed GPU Infrastructure Usage for Users Based on Docker Containers Using JupyterLab*
Advisors : 1. Ir. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.
2. Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

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 kehadirat Tuhan Yang Maha Esa yang memberikan karunia, rahmat, dan pertologan 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, diantarnya adalah:

1. Tuhan Yang Maha Esa, atas karunia dan rahmat-Nya sehingga penulis dapat mencapai titik akahir 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

DAFTAR GAMBAR

[Halaman ini sengaja dikosongkan]

DAFTAR TABEL

[Halaman ini sengaja dikosongkan]

BAB I
PENDAHULUAN

1.1 Latar Belakang

GPU telah menjadi elemen krusial dalam komputasi modern, dengan penggunaan GPU untuk deep learning workloads meningkat secara eksponensial dalam dekade terakhir. Artikel "Deep Learning Workload Scheduling in GPU Datacenters: A Survey" mengidentifikasi bahwa traditional approaches yang dirancang untuk big data atau HPC workloads tidak dapat mendukung deep learning workloads untuk fully utilize GPU resources, sehingga memerlukan pendekatan scheduling yang khusus dirancang untuk karakteristik unik dari AI workloads.

Teknologi seperti Docker Container telah menjadi solusi inovatif untuk meningkatkan efisiensi dalam pengelolaan aplikasi, terutama di lingkungan komputasi modern. Dengan mengisolasi aplikasi dan dependensinya, *container* memungkinkan penyebaran yang cepat dan konsisten. Artikel "*Containerisation for High Performance Computing Systems: Survey and Prospects*" menjelaskan bahwa teknologi kontainerisasi tidak hanya relevan dalam komputasi awan, tetapi juga memiliki potensi besar untuk sistem *High Performance Computing (HPC)*. Dalam konteks *HPC*, kontainer mampu mengemas pustaka yang dioptimalkan untuk perangkat keras tertentu, meskipun tantangan seperti ukuran yang besar dan kebutuhan orkestrasi yang kompleks tetap perlu diatasi. Penggunaan mekanisme orkestrasi yang efisien dapat membantu memaksimalkan pemanfaatan sumber daya GPU dalam lingkungan *HPC* yang terdistribusi.

Namun, tantangan seperti ukuran kontainer yang besar dan kebutuhan akan mekanisme orkestrasi yang kompleks tetap perlu diatasi. Artikel ini juga menyoroti bahwa penggunaan teknologi orkestrasi yang efisien, seperti Kubernetes, dapat membantu memaksimalkan pemanfaatan sumber daya GPU dalam lingkungan *HPC* yang terdistribusi, menjadikan kontainerisasi solusi yang menjanjikan untuk pengelolaan sumber daya multi-pengguna (Zhou et al., 2022).

Integrasi teknologi kontainer seperti Docker dengan JupyterLab telah membuka peluang baru dalam pengelolaan infrastruktur komputasi berbasis GPU untuk proyek kecerdasan buatan (AI). Artikel "*An accessible infrastructure for artificial intelligence using a Docker-based JupyterLab in Galaxy*" menunjukkan bahwa pendekatan berbasis kontainer ini dapat menyediakan lingkungan komputasi yang terisolasi namun fleksibel, memungkinkan *training model deep learning* yang cepat dan aman melalui akses GPU yang teroptimalkan. Selain itu, JupyterLab yang berjalan di atas Docker mendukung kolaborasi antar peneliti melalui *notebook* interaktif yang mudah diakses. Infrastruktur ini tidak hanya meningkatkan efisiensi penggunaan GPU tetapi juga memungkinkan pengelolaan sumber daya secara lebih adil, sebagaimana diilustrasikan dalam studi kasus pelatihan model untuk analisis gambar dan prediksi struktur protein. Dengan menggunakan pendekatan serupa, penelitian ini bertujuan untuk mengembangkan mekanisme penjadwalan GPU yang efektif dalam lingkungan terdistribusi, guna mendukung kebutuhan komputasi AI modern yang semakin kompleks dan kolaboratif.

Mekanisme penjadwalan pengguna pada lingkungan GPU terdistribusi menjadi aspek krusial dalam mendukung efisiensi dan keadilan alokasi sumber daya. Dalam konteks ini, JupyterLab menawarkan antarmuka interaktif yang memungkinkan pengguna menjalankan tugas se-

cara bersamaan dengan akses GPU yang terorkestrasi oleh Docker Container. Hal ini relevan dalam skenario penelitian atau pengembangan model AI oleh kami yang memanfaatkan sumber daya GPU secara kolektif. Dengan demikian, penelitian ini bertujuan untuk mengembangkan solusi penjadwalan yang tidak hanya meningkatkan efisiensi, tetapi juga memaksimalkan pengalaman pengguna dalam mengakses sumber daya GPU pada klaster terdistribusi.

1.2 Rumusan Masalah

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

1. Bagaimana cara mengelola penggunaan GPU agar dapat digunakan secara adil dan efisien oleh banyak pengguna?
2. Bagaimana cara memanfaatkan kontainerisasi untuk mengisolasi lingkungan kerja setiap pengguna dan meningkatkan efisiensi serta skalabilitas?
3. Apakah sistem yang diusulkan dapat mengakomodasi kebutuhan penggunaan GPU oleh banyak pengguna secara bersamaan?
4. Bagaimana cara mempermudah pengguna dalam mengakses dan memanfaatkan GPU melalui antarmuka yang interaktif?

1.3 Batasan Masalah atau Ruang Lingkup

Batasan dalam pengerjaan tugas akhir ini adalah sebagai berikut:

1. Infrastruktur GPU yang digunakan adalah infrastruktur berbasis klaster komputer yang tersedia pada laboratorium atau institusi pendidikan, dengan konfigurasi spesifik seperti *node* berbasis Docker.
2. Implementasi sistem difokuskan pada integrasi Docker dengan JupyterLab untuk orkestrasi akses GPU.

1.4 Tujuan

Tujuan dari pembuatan Tugas Akhir ini adalah sebagai berikut:

1. Mengembangkan sistem yang mampu meningkatkan efisiensi penggunaan GPU, khususnya dalam konteks lingkungan multi-pengguna.
2. Menyediakan antarmuka berbasis JupyterLab yang memungkinkan akses GPU dengan mudah, interaktif, dan terintegrasi dengan baik.

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. Sistem ini memberikan akses GPU yang lebih mudah, terstruktur, dan aman, sehingga mendukung produktivitas dalam pengembangan aplikasi berbasis GPU.

1.6 Sistematika Penulisan

Laporan penelitian tugas akhir ini terbagi menjadi Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. yaitu:

1. **BAB I Pendahuluan**
   Bab ini berisi latar belakang penilitian yang menjelaskan pentingnya pengelolaan infrastruktur GPU terdistribusi. rumusan malasah yang dihadapai dalam penggunaaan GPU multi-user, batasan masalah dan ruang lingkup penelitian, tujuan yang ingin dicapai, manfaat penelitian, serta sistematikan penulisan laporan.

2. **BAB II Tinjauan Pustaka**
   Bab ini berisi tinjauan terhadapa penelitian-penelitian terdahulu yang relevan dengan topik penelitian, teori dan konsep dasar yang meliputi klaster 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 membahs peraltan 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 terdahap 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 klaster 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 klaster 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 klaster 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 klaster 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 klaster 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 Klaster 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:

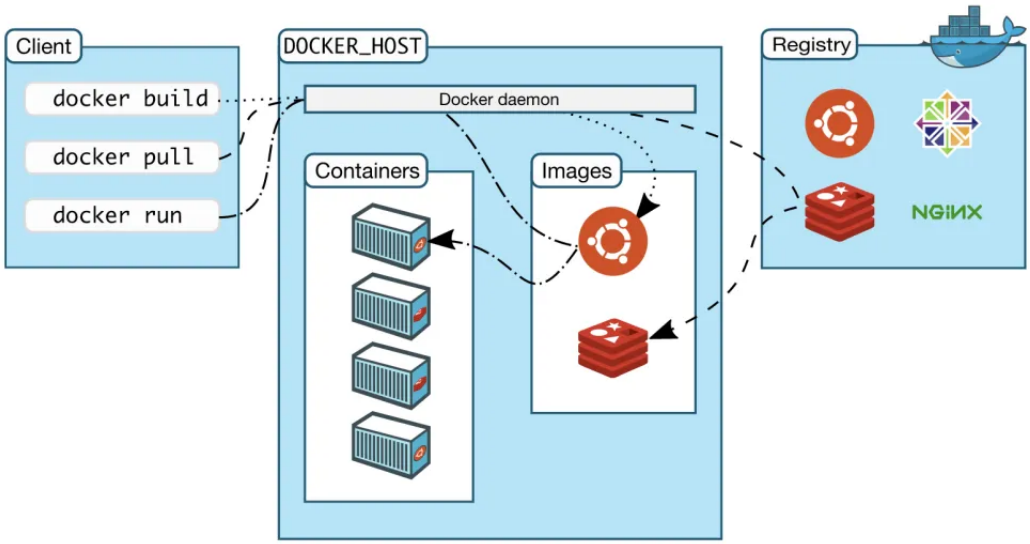
- Docker Client
  Docker Client adalah antarmuka utama yang digunakan oleh pengguna untuk berinteraksi dengan Docker. Melalui Docker Client, pengguna dapat mengirim perintah seperti membangun, mendistribusikan, dan menjalankan *container*. Perintah-perintah ini kemudian diteruskan ke Docker Daemon untuk 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 di*update* 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 Penjadwalan GPU dengan Ray

Penjadwalan GPU merupakan komponen penting dalam sistem komputasi terdistribusi, terutama ketika sumber daya GPU terbatas harus dibagi ke banyak pengguna atau tugas. Ray menyediakan mekanisme penjadwalan tugas berbasis sumber daya yang fleksibel dan dinamis untuk mengelola alokasi GPU secara efisien dalam lingkungan multi-pengguna.

2.2.4 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, *termina*l, 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 container Docker terisolasi, 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 container Docker, 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:** SAkses 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.

2.2.5 JupyterHub

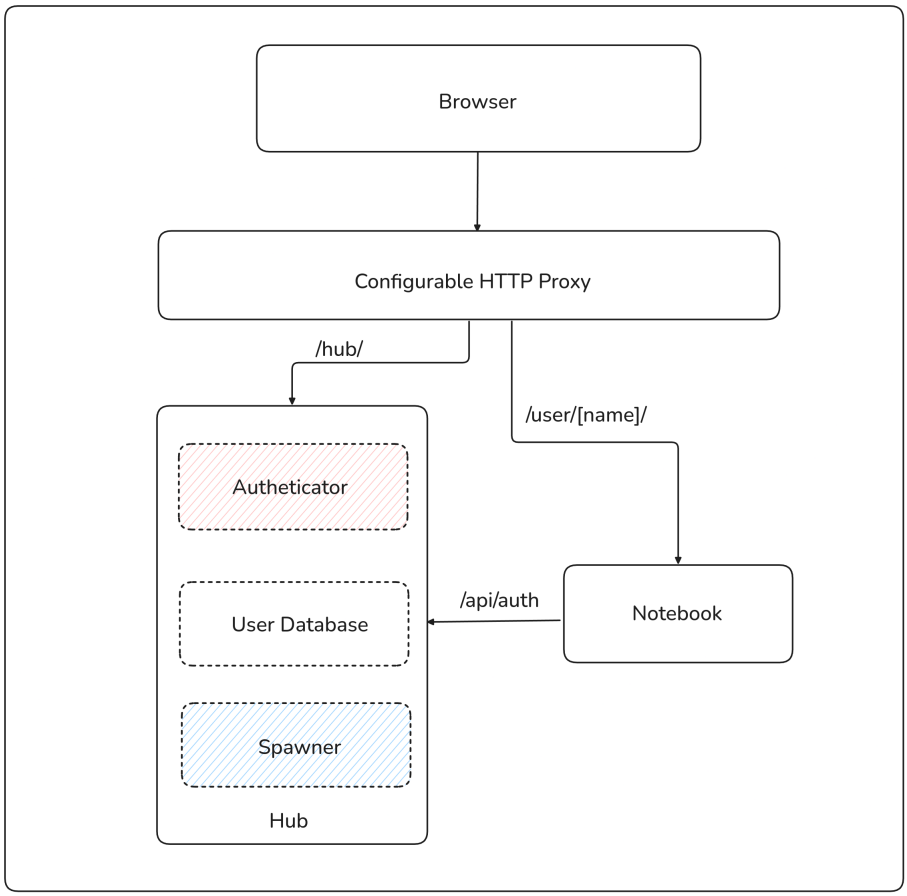
JupyterHub adalah platform open-source yang memungkinkan banyak pengguna untuk mengakses dan menjalankan lingkungan Jupyter Notebook secara terisolasi melalui antarmuka web. Dirancang untuk mendukung skenario multi-pengguna, JupyterHub sangat cocok digunakan dalam lingkungan pendidikan, penelitian, dan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

B. 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
)

Alur Kerja JupyterHub
Proses interaksi pengguna dengan JupyterHub dapat dijelaskan sebagai berikut:

1. Akses Awal: Pengguna mengakses JupyterHub melalui browser web dengan mengunjungi alamat IP atau nama domain yang telah dikonfigurasi.

2. Autentikasi: Data login yang dimasukkan oleh pengguna dikirim ke komponen Authenticator untuk validasi. Jika valid, pengguna akan dikenali dan diizinkan untuk melanjutkan.

3. Peluncuran Server Notebook: Setelah autentikasi berhasil, JupyterHub akan meluncurkan instance 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.

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 manajemene penggunakn seacara 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 klaster.
  - **MultiNodeSpawner (kustom spawner** Turunan dari **DockerSpawner** yang telah dimodifikasi untuk mendukung pemilihan npde 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.

2.2.6 Ray Framework

Ray merupakan sebuah *framework open-source* yang dirancang untuk membangun dan menjalankan aplikasi komputasi paralel dan terdistribusi secara efisien. Framework ini menyediakan abstraksi tingkat tinggi yang memungkinkan pengembang untuk membuat aplikasi yang skalabel dan mudah dijalankan pada klaster komputasi yang terdiri dari banyak node (Moritz et al., 2018).

Ray mendukung dua paradigma pemrograman utama:

1. Task-based Computing (Stateless)
Task-based computing memungkinkan pengguna untuk menjalankan fungsi Python secara paralel menggunakan dekorator `@ray.remote`. Model ini bersifat stateless dan cocok digunakan untuk proses komputasi yang dapat dibagi menjadi unit-unit kecil independen.

2. Actor-based Computing (Stateful)
Paradigma ini memungkinkan pengguna untuk membuat komponen yang mempertahankan state selama siklus hidupnya. Cocok untuk aplikasi dengan shared state atau layanan yang berjalan terus-menerus.

Arsitektur Ray
Arsitektur Ray terbagi menjadi dua lapisan utama, yaitu **Application Layer** dan **System Layer (Backend)**.

Application Layer
Pada lapisan aplikasi, terdapat beberapa komponen utama:

- **Driver**: Komponen yang memulai eksekusi program Ray dan bertanggung jawab dalam mengatur tasks dan actors.
- **Worker**: Unit eksekusi stateless yang digunakan untuk menjalankan fungsi-fungsi remote.
- **Actor**: Unit eksekusi stateful yang mempertahankan data selama proses berjalan.

System Layer (Backend)
Lapisan sistem ini mencakup manajemen koordinasi, scheduling, dan penyimpanan objek:

1. Object Store Object Store adalah penyimpanan berbasis memori untuk menyimpan objek hasil eksekusi:

- Mendukung *zero-copy* antar proses
- Manajemen memori otomatis
- Penyimpanan efisien untuk objek besar

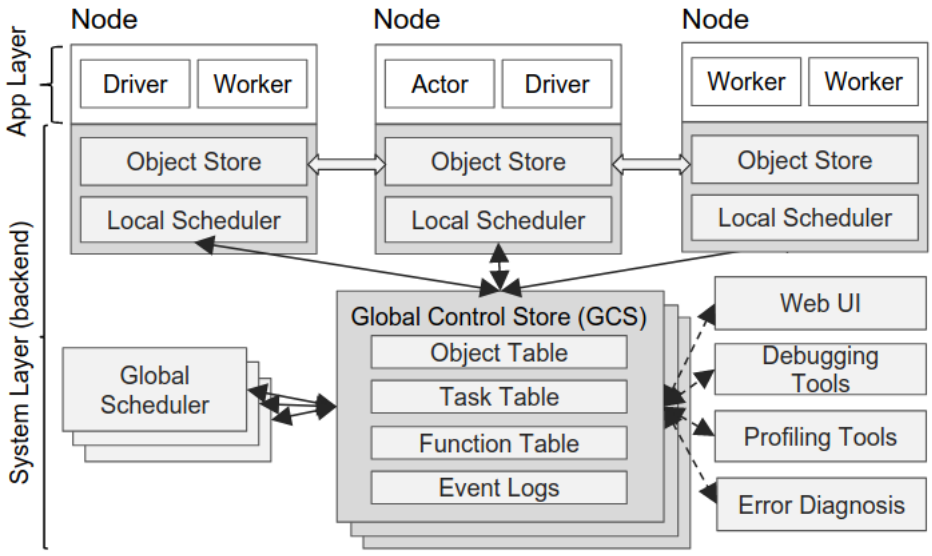
2. Raylet (Local Scheduler + Object Manager)

- Menangani penjadwalan tasks lokal
- Mengelola transfer objek antar node
- Berinteraksi dengan Global Control Store (GCS)

3. Global Control Store (GCS) Komponen pusat metadata dan koordinasi antar node:

- Menyimpan metadata objek, status tasks, actors, dan nodes
- Menyediakan event logs untuk debugging

4. Global Scheduler Mengatur penjadwalan global saat node lokal tidak memiliki resource yang mencukupi.



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

[Halaman ini sengaja dikosongkan]

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.

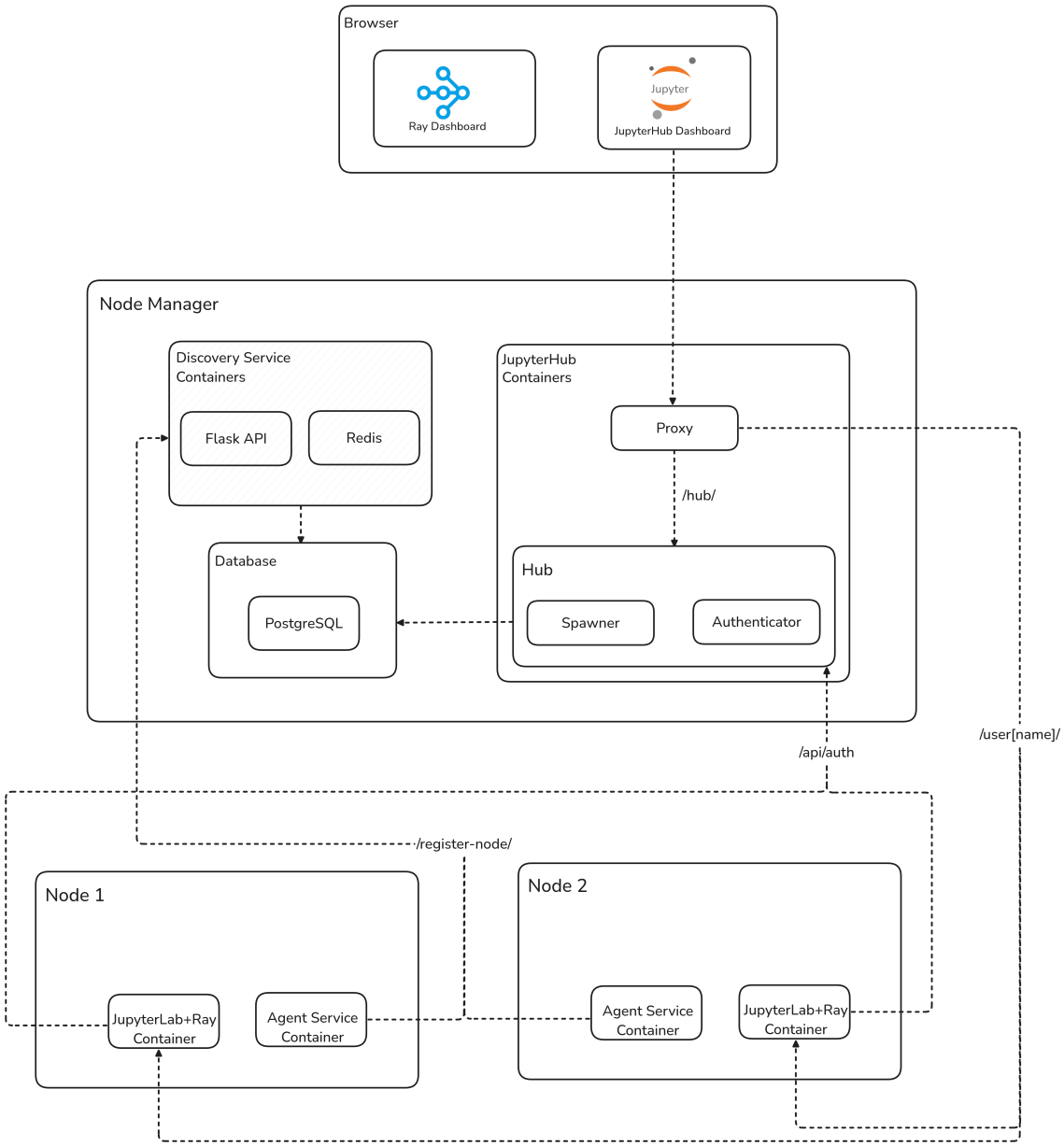
Langkah awal dalam perancangan adalah merancang sistem service discovery yang berfungsi sebagai pusat pengumpulan dan penyimpanan status sumber daya dari seluruh node yang tersedia dalam klaster. Data ini mencakup informasi penggunaan CPU, RAM, keberadaan GPU, serta jumlah container aktif, yang dikirim 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 klaster. 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 klaster. 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 Discovery Service

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.

```
from flask import Flask, jsonify
from flask_cors import CORS
from flask_migrate import Migrate

# Import configuration
from config import Config

# Import models and database
from models import db

# Import blueprints
from routes.node_routes import node_bp
from routes.profile_routes import profile_bp

import logging
import os

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("DiscoveryAPI")

def create_app():
    """Application factory pattern"""
    app = Flask(__name__)

    # Load configuration
    app.config.from_object(Config)

    # Initialize extensions
    CORS(app, origins="*")
    db.init_app(app)
    Migrate(app, db)

    # Register blueprints
    app.register_blueprint(node_bp, url_prefix='')
    app.register_blueprint(profile_bp, url_prefix='')

    # Health check route
    @app.route("/health-check")
    def health_check():
        from services.redis_service import RedisService
        redis_service = RedisService()

        return jsonify({
```

```
            "status": "ok",
            "message": "Hello, from [DiscoveryAPI]",
            "database": {
                "postgres": "connected" if db.engine else "disconnected",
                "redis": "connected" if redis_service.is_connected() else↩
                    "disconnected"
            },
            # "config": {
            #     "redis_host": Config.REDIS_HOST,
            #     "redis_port": Config.REDIS_PORT,
            #     "postgres_host": Config.POSTGRES_HOST,
            #     "postgres_port": Config.POSTGRES_PORT
            # }
        }), 200


    with app.app_context():
        db.create_all()

        # Initialize default profiles
        from services.profile_service import ProfileService
        try:
            ProfileService.create_default_profiles()
            logger.info("Default profiles initialized")
        except Exception as e:
            logger.error(f"Error initializing default profiles: {e}")

    return app

def run_periodic_tasks(app):
    """Run periodic tasks"""
    import threading
    import time

    def cleanup_inactive_nodes():
        with app.app_context():
            from services.node_service import NodeService
            from services.redis_service import RedisService

            redis_service = RedisService()
            node_service = NodeService(redis_service)

            while True:
                try:
                    node_service.mark_nodes_inactive()
                    logger.info("Cleaned up inactive nodes")
                except Exception as e:
                    logger.error(f"Error in cleanup task: {e}")

                time.sleep(300)

    # Start cleanup thread
    cleanup_thread = threading.Thread(target=cleanup_inactive_nodes, ↩
        daemon=True)
    cleanup_thread.start()

if __name__ == '__main__':
```

```
    app = create_app()

    # Start periodic tasks
    run_periodic_tasks(app)

    # Run the application
    app.run(debug=True, host='0.0.0.0', port=15002)
```

Listing 3.1: Potongan Kode File *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`.

```
class Node(db.Model):
    __tablename__ = 'nodes'

    id = db.Column(db.Integer, primary_key=True)
    hostname = db.Column(db.String(255), unique=True, nullable=False)
    ip = db.Column(db.String(45), nullable=False)
    cpu_cores = db.Column(db.Integer, nullable=False)
    ram_gb = db.Column(db.Float, nullable=False)
    has_gpu = db.Column(db.Boolean, default=False)
    gpu_info = db.Column(JSON, default=list)

    is_active = db.Column(db.Boolean, default=True)
    max_containers = db.Column(db.Integer, default=10)
    last_heartbeat = db.Column(db.DateTime)
    ...
```

Listing 3.2: Potongan Kode Model `Node`

```python
class Profile(db.Model):
    __tablename__ = 'profiles'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True, nullable=False)
    cpu_requirement = db.Column(db.Integer)
    ram_requirement = db.Column(db.Float)
    gpu_required = db.Column(db.Boolean, default=False)
    max_cpu_usage = db.Column(db.Float, default=80.0)
    max_memory_usage = db.Column(db.Float, default=85.0)
    ...
```

Listing 3.3: Potongan Kode Model `Profile`


```python
class Profile(db.Model):
    __tablename__ = 'profiles'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True, nullable=False)
    cpu_requirement = db.Column(db.Integer)
    ram_requirement = db.Column(db.Float)
    gpu_required = db.Column(db.Boolean, default=False)
    max_cpu_usage = db.Column(db.Float, default=80.0)
    max_memory_usage = db.Column(db.Float, default=85.0)
    ...
```


Listing 3.4: Potongan Kode Model `Profile`

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/disk usage.
- Menentukan apakah node masih aktif berdasarkan heartbeat agent.

Koneksi ke Redis dilakukan melalui file `redis_client.py` dan kelas `RedisService`. Semua konfigurasi basis data diatur melalui file `config.py` yang memuat environment variable seperti `POSTGRES_HOST`, `REDIS_PORT`, dan `REDIS_PASSWORD`.

D. Seleksi dan Load Balancing 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.

```
def calculate_node_score(node_data: dict) -> float:
    cpu_usage = node_data.get("cpu_usage_percent", 100)
    memory_usage = node_data.get("memory_usage_percent", 100)
    score = (cpu_usage * Config.CPU_WEIGHT) + (memory_usage * Config.↩
       MEMORY_WEIGHT)

    if cpu_usage > 90 or memory_usage > 90:
        score += Config.HEAVY_PENALTY
    elif cpu_usage > 80 or memory_usage > 80:
        score += Config.MEDIUM_PENALTY

    return round(score, 2)
```

Listing 3.5: Fungsi Perhitungan Skor Node

Pseudocode pada Algoritma **??** menggambarkan proses perhitungan skor beban berdasarkan kombinasi penggunaan CPU dan memori, serta pemberian penalti jika suatu node berada pada kondisi kelebihan beban. Skor ini digunakan sebagai dasar dalam pemilihan node untuk menjalankan layanan JupyterLab secara efisien.

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 Environtment

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

```
class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY', 'secret-service-1111')
    DEBUG = os.environ.get('DEBUG', 'True').lower() == 'true'

    # Database Configuration
    POSTGRES_HOST = os.environ.get('POSTGRES_HOST', 'localhost')
    POSTGRES_PORT = os.environ.get('POSTGRES_PORT', '5432')
    POSTGRES_DB = os.environ.get('POSTGRES_DB', 'discovery')
    POSTGRES_USER = os.environ.get('POSTGRES_USER', 'postgres')
    POSTGRES_PASSWORD = os.environ.get('POSTGRES_PASSWORD', 'postgres')
```

```
11
12      SQLALCHEMY_DATABASE_URI = (
13          f"postgresql://{POSTGRES_USER}:{POSTGRES_PASSWORD}@"
14          f"{POSTGRES_HOST}:{POSTGRES_PORT}/{POSTGRES_DB}"
15      )
16
17      SQLALCHEMY_TRACK_MODIFICATIONS = False
18      SQLALCHEMY_ECHO = os.environ.get('SQLALCHEMY_ECHO', 'false').lower() ↩
            == 'true'
19
20      # Redis
21      REDIS_HOST = os.environ.get('REDIS_HOST', 'localhost')
22      REDIS_PORT = int(os.environ.get('REDIS_PORT', 6379))
23      REDIS_PASSWORD = os.environ.get('REDIS_PASSWORD', 'redis@pass')
24      REDIS_EXPIRE_SECONDS = int(os.environ.get('REDIS_EXPIRE_SECONDS', 45)↩
            )
25
26      # Load Balancer
27      DEFAULT_MAX_CPU_USAGE = 80.0
28      DEFAULT_MAX_MEMORY_USAGE = 85.0
29      STRICT_MAX_CPU_USAGE = 60.0
30      STRICT_MAX_MEMORY_USAGE = 60.0
31      STRICT_MAX_CONTAINERS = 5
32
33      # Skoring
34      CPU_WEIGHT = 0.8
35      MEMORY_WEIGHT = 0.8
36      HEAVY_PENALTY = 80
37      MEDIUM_PENALTY = 20
```

Listing 3.6: Potongan Kode File 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 3.2 berikut:

Tabel 3.2: 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=16379`	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.

```
# syntax=docker/dockerfile:1.3

FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    TZ=Asia/Jakarta

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 15002
CMD ["python", "app.py"]
```

Listing 3.7: Dockerfile Service Discovery

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

```
version: "3.8"

services:
  discovery:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: discovery-api
    restart: unless-stopped
    ports:
      - "15002:15002"
    environment:
      POSTGRES_HOST: postgres
      POSTGRES_PORT: 5432
      POSTGRES_DB: 'voyager'
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres

      REDIS_HOST: redis
      REDIS_PORT: 16379
      REDIS_PASSWORD: "redis@pass"
      REDIS_EXPIRE_SECONDS: 45

      API_HOST: 0.0.0.0
      API_PORT: 15002
```

```
      DEBUG: "True"
      SECRET_KEY: "secret-service111111"
    depends_on:
      - postgres
      - redis
    networks:
      - discovery-network

  postgres:
    image: postgres:14-alpine
    container_name: postgres
    restart: unless-stopped
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: voyager
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_INITDB_ARGS: "--encoding=UTF-8"
      TZ: Asia/Jakarta
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres -d voyager"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s
    networks:
      - discovery-network

  redis:
    image: redis:7-alpine
    container_name: redis
    restart: unless-stopped
    volumes:
      - redis_data:/data
      - ./redis.conf:/usr/local/etc/redis/redis.conf
    command: ["redis-server", "/usr/local/etc/redis/redis.conf"]
    environment:
      TZ: Asia/Jakarta
    ports:
      - "16379:16379"
    healthcheck:
      test: ["CMD", "redis-cli", "-p", "16379", "ping"]
      interval: 10s
      timeout: 3s
      retries: 3
    networks:
      - discovery-network

volumes:
  postgres_data:
  redis_data:

networks:
```

```
83   discovery-network:
84     driver: bridge
```

Listing 3.8: Docker Compose Discovery Service

`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
```

Listing 3.9: Menjalankan Discovery Service via Docker Compose

Perintah tersebut akan:

- Membuild image `discovery` dari Dockerfile.
- Menjalankan container PostgreSQL dan Redis terlebih dahulu melalui `depends_on`.
- Menyediakan API Discovery di `http://localhost:15002`.

G. List API Endpoint

Tabel 3.3: 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 Agent Service

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.

bol

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 3.10. Fungsi ini bertugas mengumpulkan data menggunakan `collect_node_info()` dan mengirimkannya ke API.

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

Listing 3.10: 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 atau AMD).

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

Listing 3.11: Kumpulan Informasi Sistem oleh Agent

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

```
def get_container_info():
    containers = docker_client.containers.list()
    for container in containers:
        if "jupyter" in container.name or "jupyter" in container.image.↩
            tags:
            ...
```

Listing 3.12: 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.

```
def get_gpu_stats():
    stats = gpustat.GPUStatCollection.new_query()
```

```
    for gpu in stats.gpus:
        gpu_info.append({
            "name": gpu.name,
            "memory_used_mb": gpu.memory_used,
            "utilization_gpu_percent": gpu.utilization
        })
```

Listing 3.13: 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.

```
if __name__ == "__main__":
    while True:
        register()
        time.sleep(15)
```

Listing 3.14: Loop Registrasi Agent Tiap 15 Detik

C. Deployment dan Kontainerisasi 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. Listing 3.15 menunjukkan isi file `Dockerfile` yang digunakan untuk membangun image Agent.

```
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    TZ=Asia/Jakarta

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 15002
CMD ["python", "agent.py"]
```

Listing 3.15: Dockerfile untuk Agent Service

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

Proses build dan run Agent dapat dilakukan melalui perintah:

```
# Build image agent
docker build -t agent-service .

# Jalankan agent di node
docker run -d --name node-agent \
  --restart always \
  --net=host \
  --env-file ./env-agent \
  agent-service
```

Listing 3.16: Perintah untuk Build dan Menjalankan Agent

File `env-agent` berisi konfigurasi lingkungan seperti alamat Discovery Service:

```
DISCOVERY_URL=http://192.168.122.1:15002/register-node
```

Listing 3.17: Contoh File .env untuk Agent

Penggunaan mode `-net=host` memungkinkan agent mengakses informasi IP node dengan benar serta membaca container aktif melalui Docker daemon lokal.

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.

Bagian ini akan membahas implementasi JupyterHub, mulai dari konfigurasi autentikasi, integrasi multi-node dengan spawner kustom, hingga alur spawn container pengguna.

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 3.4: Spesifikasi Peralatan Pendukung

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

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

Tabel 3.5: Daftar Perangkat Lunak Pendukung

Nama Perangkat Lunak	**Versi**	**Keterangan**
Docker Engine	24.x	Digunakan untuk menjalankan container JupyterLab dan Ray Worker secara terisolasi. Memungkinkan lingkungan komputasi tiap pengguna berjalan secara independen dan mudah didistribusikan ke berbagai node.
Docker Compose	2.x	Membantu mendefinisikan dan mengatur layanan multi-container (seperti JupyterHub dan Redis) 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	Bahasa pemrograman utama yang digunakan untuk seluruh komponen sistem, seperti konfigurasi JupyterHub, pengembangan REST API, Ray, serta skrip monitoring.

[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 Skenario Pengujian

Pengujian dilakukan dengan lima skenario yang dirancang untuk mewakili berbagai kondisi nyata dalam penggunaan sistem. Setiap skenario difokuskan pada aspek tertentu seperti seleksi node, distribusi user, validasi profil GPU, serta penanganan beban dan kegagalan node.

4.1.1 Skenario 1: Pemilihan Node dengan Beban Terendah

- **Tujuan:** Memastikan sistem memilih node dengan CPU usage terendah.
- **Langkah:** Jalankan 1 user (profil: `single-cpu`) dan catat node terpilih.
- **Hasil:** Sistem memilih node dengan 24.7% CPU, lebih rendah dari kandidat lainnya.

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.

4.1.3 Skenario 3: Simulasi Beban Tinggi

- **Tujuan:** Memastikan sistem menghindari node yang sedang overload.
- **Langkah:** Menjalankan `stress-ng` di satu node.
- **Hasil:** Node tersebut tidak dipilih oleh sistem.

4.1.4 Skenario 4: Validasi Profil GPU

- **Tujuan:** User GPU hanya boleh dialokasikan ke node dengan GPU.
- **Hasil:** Semua user GPU dialokasikan ke node dengan NVIDIA A100.

4.1.5 Skenario 5: TTL Redis dan Node Tidak Aktif

- **Tujuan:** Node yang tidak update status akan dihapus otomatis.
- **Langkah:** Matikan agent dan tunggu TTL (45s), lalu spawn user.
- **Hasil:** Node tidak aktif diabaikan oleh Discovery API.

4.2 Evaluasi Pengujian

Tabel berikut merangkum hasil pengujian dan status keberhasilan sistem dalam setiap skenario:

Tabel 4.1: Ringkasan Evaluasi Pengujian Sistem

No	**>**Skenario Pengujian	**Berhasil**
1	Pemilihan node dengan CPU terendah untuk profil `single-cpu`	Ya
2	Distribusi 5 user secara paralel (profil campuran CPU dan GPU)	Ya
3	Sistem menghindari node yang overload (stress test)	Ya
4	Alokasi profil GPU hanya ke node yang memiliki GPU	Ya
5	Node dengan agent mati tidak terpilih setelah TTL Redis habis	Ya

Evaluasi menunjukkan bahwa sistem berhasil melakukan alokasi kontainer sesuai dengan desain arsitektur dan kriteria seleksi node. Load balancing bekerja optimal, dan fitur failover serta integrasi antar komponen telah diuji dengan baik.

BAB V
PENUTUP

5.1 Kesimpulan

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

1. Pembuatan 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.

5.2 Saran

Untuk pengembangan lebih lanjut pada Lorem ipsum dolor sit amet, consectetuer 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 pada Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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. 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. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

[Halaman ini sengaja dikosongkan]