

Penerapan Algoritma *Greedy* Untuk Optimasi Urutan Instruksi Dockerfile dalam Memaksimalkan *Cache Hit* Pada *Pipeline* CI/CD

Fazri Arrashyi Putra - 13524127

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: fazriarrashyiputra@gmail.com , 13524127@std.stei.itb.ac.id

Abstract—Proses build Docker dalam *pipeline* CI/CD sangat bergantung pada mekanisme layer caching untuk mempercepat waktu kompilasi, namun penulisan instruksi Dockerfile yang kurang terstruktur sering memicu invalidasi cache beruntun di mana perubahan kecil di layer atas merusak cache di bawahnya, sehingga sistem membuang waktu untuk mengkompilasi ulang dependensi yang sebenarnya tidak berubah. Untuk mengatasi masalah tersebut, makalah ini mengusulkan metode penyusunan ulang instruksi Dockerfile secara otomatis menggunakan algoritma *Greedy*. Dalam penerapannya, setiap baris instruksi dipetakan terlebih dahulu dalam bentuk *Dependency Graph* untuk memastikan agar ketergantungan antar kode tetap aman dan tidak error saat build. Setelah itu, algoritma *Greedy* akan menyusun ulang urutan berdasarkan tingkat volatilitasnya. Instruksi yang jarang berubah ditempatkan di layer teratas, sementara instruksi yang lebih dinamis digeser ke posisi terbawah. Pendekatan heuristik yang praktis ini efektif mencegah hilangnya cache pada proses instalasi yang berat, yang akhirnya secara signifikan mampu meningkatkan persentase cache hit dan memangkas total durasi build image secara keseluruhan.

Keywords—*Dockerfile*; *Cache Invalidation*; *Algoritma Greedy*; *Dependency Graph*; *CI/CD*

I. PENDAHULUAN

Dalam siklus pengembangan perangkat lunak modern, praktik *Continuous Integration* dan *Continuous Deployment* (CI/CD) telah menjadi standar industri. Setiap kali programmer melakukan commit atau mengunggah kode baru ke repositori, *pipeline* CI/CD akan secara otomatis menjalankan proses kompilasi (build), pengujian (testing), hingga peluncuran (*deployment*). Di sebagian besar arsitektur saat ini, proses build tersebut sangat bergantung pada teknologi kontainerisasi, khususnya Docker.

Docker menggunakan sistem layer sistem caching untuk mempercepat proses pembuatan image. Setiap instruksi di dalam Dockerfile (seperti FROM, RUN, COPY) akan menciptakan layer baru. Jika instruksi tersebut tidak berubah pada build berikutnya, Docker akan menggunakan ulang (reuse) layer yang sudah ada di memori cache, sehingga prosesnya bisa selesai dalam hitungan detik [1], [2],

Namun, permasalahan klasik sering muncul akibat kelalaian pengembang dalam menyusun urutan instruksi. Mekanisme cache Docker bersifat sekuensial dan rentan terhadap efek domino (invalidasi cache beruntun). Artinya, jika satu layer di posisi atas mengalami perubahan, maka seluruh layer di bawahnya akan otomatis hilang dan harus dieksekusi ulang dari nol. Akibatnya, instruksi berat seperti mengunduh dependensi yang memakan waktu ber menit-menit harus diulang terus-menerus, membuang sumber daya dan memperlambat *pipeline* kerja tim.

Untuk menyelesaikan masalah ini, penyusunan urutan Dockerfile tidak bisa dilakukan secara manual jika aplikasinya sudah kompleks. Oleh karena itu, makalah ini mengeksplorasi penerapan algoritma *greedy* berbasis *dependency graph* untuk mengotomatisasi penyusunan ulang instruksi Dockerfile.

II. LANDASAN TEORI

A. Arsitektur Layer Caching pada Docker

Setiap instruksi dalam Dockerfile tidak dieksekusi sebagai satu kesatuan, melainkan diterjemahkan menjadi tumpukan layer *image*. Saat perintah docker build dijalankan daemon Docker akan mengevaluasi setiap instruksi satu per satu.

Mari kita tinjau contoh kode Dockerfile yang buruk.

```
# Contoh Dockerfile yang tidak optimal
FROM node:18-alpine
WORKDIR /app

# Meng-copy SELURUH file (termasuk kode
yang sering berubah) di awal
COPY . .

# Menginstal dependensi (proses berat dan
memakan waktu)
RUN npm install

# Build aplikasi
RUN npm run build
EXPOSE 3000
```

```
CMD ["npm", "start"]
```

Pada kode di atas, COPY . . dipanggil sebelum npm install. Jika programmer hanya mengubah satu baris teks pada file misalkan index.html, instruksi COPY . . akan mendeteksi adanya perubahan hash file. Akibatnya, layer COPY tersebut akan membuat *cache miss*. Buruknya, Docker akan langsung menginvalidasi semua cache di bawahnya [1], memaksa server untuk menjalankan RUN npm install dari awal untuk mengunduh ratusan MB paket, padahal tidak ada dependensi yang bertambah.

Konsep optimasi dasar mengharuskan kita memisahkan instruksi yang statis (jarang berubah) dan menempatkannya di atas instruksi yang dinamis (sering berubah), selama hal itu tidak menyalahi aturan ketergantungan [1].

B. Directed Acyclic Graph (DAG) pada Instruksi

Untuk dapat diubah urutannya oleh algoritma, instruksi-instruksi tidak bisa asal dipindah. Misalnya, kita tidak bisa memindahkan RUN npm install ke atas WORKDIR /app jika install tersebut membutuhkan direktori kerja spesifik.

Oleh karena itu, instruksi Dockerfile dipetakan ke dalam bentuk Directed Acyclic Graph (DAG). DAG adalah jenis graf berarah yang tidak memiliki siklus [3]. Setiap simpul merepresentasikan satu baris instruksi, dan edge merepresentasikan ketergantungan antar instruksi.

- **Node A:** COPY package.json .
- **Node B:** RUN npm install
- Edge menghubungkan A -> B, yang berarti Node A harus dieksekusi mutlak sebelum node B.

Dengan memetakan ketergantungan ini, sistem optimasi memiliki batas aman untuk memindahkan urutan instruksi tanpa menyebabkan menjadi error [3].

C. Algoritma Greedy Berbasis Volatilitas

Algoritma *Greedy* adalah pendekatan optimasi yang selalu mengambil pilihan lokal terbaik (local optimum) pada setiap langkahnya dengan harapan akan menghasilkan solusi global terbaik (global optimum) [3].

Dalam konteks penyusunan urutan Dockerfile, kita tidak bisa sekedar memindahkan instruksi secara acak. Algoritma *Greedy* membutuhkan fungsi heuristik atau bobot untuk menentukan pilihan terbaiknya. Metrik yang digunakan adalah Volatilitas (V), yakni seberapa sering sebuah layer berubah.

III. METODOLOGI DAN DESAIN ALGORITMA

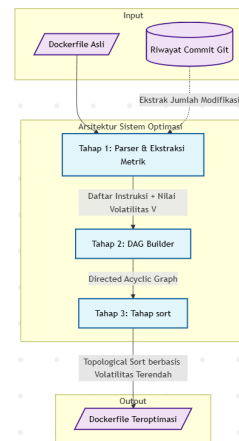
Pada bab ini, akan diuraikan secara menyeluruh mengenai metodologi dan rancangan arsitektur dari sistem optimasi Dockerfile, bagian ini berfokus pada perancangan arsitektur dari mesin optimasi itu sendiri, mulai dari fase ekstraksi

Dockerfile asli, pembentukan DAG, hingga eksekusi pencarian nilai optimal menggunakan algoritma *greedy*.

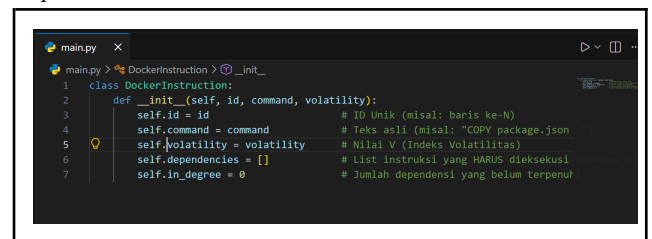
A. Arsitektur Sistem Optimasi

Sistem optimasi urutan Dockerfile yang dirancang dalam makalah ini dilakukan melalui tiga tahapan utama secara berurutan:

1. Tahap *Parsing* dan Ekstraksi Metrik: Sistem membaca Dockerfile asli yang sebelum dioptimasi baris demi baris. Pada tahap ini juga sistem membaca riwayat version system control untuk menghitung jumlah commit dari setiap file guna mendapatkan nilai indeks volatilitas.
2. Tahap pembangunan Graf (DAG *Builder*): Instruksi yang telah diekstrak dipetakan menjadi simpul. Ketergantungan logis antar instruksi dihubungkan menggunakan sisi, membentuk DAG.
3. Tahap Sorting: Algoritma *Greedy* melakukan Topological Sorting pada DAG. Pada setiap iterasinya, algoritma akan menyeleksi simpul yang memiliki nilai V terkecil untuk dimasukkan ke dalam draf Dockerfile yang baru.



B. Representasi Struktur Data



Pembentukan DAG tidak dapat dilakukan secara sembarangan. Sistem parser harus memiliki penganalisa semantik sederhana untuk mendeteksi ketergantungan antar instruksi. Sebagai contoh praktis pada containerisasi aplikasi backend modern yang menggunakan framework seperti Laravel, instruksi instalasi library memiliki hierarki yang kaku.

Aturan pembentukan sisi dalam sistem optimasi ini didefinisikan melalui tiga aturan deteksi:

1. Directed File Dependency: instruksi eksekusi library seperti RUN composer install atau RUN npm install akan selalu membentuk hubungan dari instruksi COPY yang membawa berkas manifestnya (seperti composer.json atau package.json).
2. Workspace dependency: semua instruksi modifikasi berkas akan terikat pada instruksi WORKDIR terakhir yang dideklarasikan.
3. Environment Dependency: Instruksi RUN yang membutuhkan koneksi basis data atau pengaturan aplikasi akan bergantung pada instruksi ENV atau ARG yang mendefinisikan parameter tersebut.

C. Implementasi Algoritma Greedy

Penyusunan ulang menggunakan algoritma greedy pada dasarnya adalah variasi dari algoritma Kahn's Topological Sort, yang dimodifikasi dengan fungsi heuristik pencarian minimum [3]. Pada antrean kandidat, sistem tidak mengambil instruksi secara acak (FIFO/LIFO), melainkan menggunakan priority queue berbasis metrik volatilitas. Berikut adalah implementasi inti algoritma greedy:

```

30 def optimize_dockerfile(instructions_graph):
31     optimized_dockerfile = []
32
33     # 1. Kumpulkan semua instruksi awal yang bebas dependensi (in-degree == 0)
34     ready_pool = [inst for inst in instructions_graph if inst.in_degree == 0]
35
36     while ready_pool:
37         # 2. TERAPKAN STRATEGI GREEDY
38         # Prioritas 1: Volatilitas terkecil (Ascending) -> paling statis di atas
39         # Prioritas 2 (tie-breaker): Waktu eksekusi terlama (Descending) -> amankan layer berat
40         ready_pool.sort(key=lambda x: (x.volatility, -x.exec_time))
41
42         # 3. Ambil instruksi paling optimal (index ke-0)
43         best_instruction = ready_pool.pop(0)
44         optimized_dockerfile.append(best_instruction)
45
46         # 4. Hapus node ini dari graf virtual, dan bebaskan children-nya
47         for child in best_instruction.children:
48             child.in_degree -= 1
49             # Jika child sudah tidak terikat dependensi apapun, masukkan ke antrean
50             if child.in_degree == 0:
51                 ready_pool.append(child)
52
53     # 5. Validasi Deadlock / Siklus
54     if len(optimized_dockerfile) != len(instructions_graph):
55         raise Exception("Gagal! Terdapat siklus (circular dependency) pada struktur graf.")
56
57     return optimized_dockerfile

```

Pada blok while ready_pool: algoritma terus mencari instruksi yang independen. Fungsi ready_pool.sort(key=lambda x: x.volatility) adalah representasi langsung dari persamaan $\min(V(I))$. Karena ini algoritma greedy, sistem tidak memperdulikan masa depan layer dan hanya fokus mencari layer paling statis pada saat itu juga.

D. Penelusuran eksekusi langkah demi langkah

Untuk memvalidasi kebenaran logika heuristik yang diterapkan, berikut adalah penelusuran (dry-run) kondisi himpunan antrean pada setiap iterasi ketika algoritma Greedy mengeksekusi Dockerfile masukan.

Misalkan sistem mendeteksi himpunan simpul awal dengan nilai volatilitas (V) dan waktu eksekusi (E) sebagai berikut:

- N1: FROM (V = 0.0)
- N2: WORKDIR (V = 0.0)
- N3: COPY pkg (V = 0.1, E = 0.5)

- N4: RUN install (V = 0.2, E = 60.0)
- N5: COPY src (V = 0.9, E = 2.0)
- N6: EXPOSE (V = 0.0, E = 0.1)

Iterasi 1 (Inisialisasi)

Algoritma memindai simpul dengan status in-degree = 0. Hanya N1 (FROM) yang memenuhi syarat karena merupakan akar graf utama.

- Antrean ready_pool: [N1]
- Pemilihan Greedy: Sistem menarik N1. Simpul N1 dihapus sehingga melepaskan ketergantungan untuk N2.
- Dockerfile terbentuk: [FROM]

Iterasi 2

Simpul N2 kini memiliki in-degree = 0.

- Antrean ready_pool: [N2]
- Pemilihan Greedy: Sistem menarik N2 (WORKDIR). Penghapusan N2 melepaskan beberapa ketergantungan sekaligus, yaitu N3, N5, dan N6.
- Dockerfile terbentuk: [FROM, WORKDIR]

Iterasi 3 (Solusi Heuristik Utama)

Pada tahap ini penyeleksian volatilitas mulai bekerja. Antrean kini memiliki tiga kandidat independen yang valid.

- Antrean ready_pool: [N3 (V = 0.1), N5 (V = 0.9), N6 (V = 0.0)]
- Pemilihan Greedy: Sistem mengurutkan nilai volatilitas dari yang terkecil ke terbesar. N6 (EXPOSE) dipilih karena memiliki nilai V = 0.0.
- Dockerfile terbentuk: [FROM, WORKDIR, EXPOSE]

Iterasi 4

Kandidat yang tersisa adalah N3 dan N5. Keduanya merupakan instruksi COPY, namun memiliki tingkat volatilitas yang berbeda secara signifikan berdasarkan riwayat perubahan Git.

- Antrean ready_pool: [N3 (V = 0.1), N5 (V = 0.9)]
- Pemilihan Greedy: Berbeda dengan urutan penulisan Dockerfile secara konvensional, algoritma tidak langsung memilih COPY src. Sistem secara "serakah" memilih berkas yang paling stabil terlebih dahulu, yaitu N3 (COPY pkg). Setelah N3 diproses, dependensi untuk N4 (RUN install) menjadi bebas.
- Dockerfile terbentuk: [FROM, WORKDIR, EXPOSE, COPY pkg]

Iterasi 5

- Antrean ready_pool: [N4 (V = 0.2), N5 (V = 0.9)]
- Pemilihan Greedy: Algoritma kembali mengurutkan kandidat berdasarkan volatilitas. N4 (RUN install) dipilih meskipun membutuhkan waktu eksekusi sekitar 60 detik karena tingkat volatilitasnya (V = 0.2) jauh lebih rendah dibandingkan kode sumber aplikasi (V = 0.9).
- Dockerfile terbentuk: [FROM, WORKDIR, EXPOSE, COPY pkg, RUN install]

Iterasi 6

- Antrean ready_pool: [N5 ($V = 0.9$)]
- Pemilihan Greedy: Hanya N5 yang tersisa sehingga dipilih dan dieksekusi.
- Dockerfile terbentuk: [FROM, WORKDIR, EXPOSE, COPY pkg, RUN install, COPY src]

Proses berlanjut hingga antrean ready_pool menjadi kosong dan seluruh simpul telah diproses. Urutan akhir yang dihasilkan algoritma adalah:

[FROM, WORKDIR, EXPOSE, COPY pkg, RUN install, COPY src]

Penelusuran ini menunjukkan bahwa algoritma Greedy mampu menghasilkan urutan instruksi yang memprioritaskan layer dengan volatilitas rendah untuk ditempatkan lebih awal. Dengan demikian, layer yang memiliki biaya komputasi tinggi seperti proses instalasi dependensi dapat mempertahankan cache lebih lama dan tidak perlu dibangun ulang setiap kali terjadi perubahan pada source code. Pendekatan ini memungkinkan optimasi cache Docker dilakukan tanpa perlu mengevaluasi seluruh kemungkinan.

E. Kompleksitas Algoritma

Kompleksitas waktu dari algoritma optimasi di atas bergantung pada proses Topological Sort dan pengurutan

1. Mengunjungi seluruh simpul dan membuat sisi memakan $O(V_{\text{simpul}} + E)$ di mana V simpul adalah jumlah instruksi dan E adalah jumlah dari ketergantungan
2. Pencarian nilai minimum pada antrean ready_pool memakan waktu $O(V \log V)$ jika menggunakan struktur min-heap atau priority queue.

Sehingga kompleksitas waktu keseluruhan adalah $O(V_{\text{simpul}} \log V_{\text{simpul}} + E)$. Mengingat panjang baris Dockerfile di industri umumnya bersikar antara 10 hingga 100 baris kode, kompleksitas asimtotik ini sangat ringan dan dapat dieksekusi secara instan oleh server CI/CD tanpa menambah beban komputasi.

IV. HASIL EKSPERIMEN DAN PEMBAHASAN

A. Lingkungan Eksperimen dan Parameter Pengujian

Untuk memvalidasi efektivitas algoritma greedy yang telah dirancang, sebuah simulasi kompilasi image dilakukan menggunakan studi kasus aplikasi berbasis node.js. Pengujian dilakukan pada lingkungan berspesifikasi VPS standar CI/CD.

Parameter yang diukur dalam eksperimen ini adalah durasi kompilasi ulang dan rasio cache hit. Pengujian disimulasikan dalam dua kondisi:

- 1) *Initial Build*: Kompilasi pertama kali saat memori cache masih kosong.
- 2) *Incremental Rebuild*: Kompilasi ulang yang dipicu oleh skenario perubahan satu baris kode pada berkas source code, untuk mensimulasikan aktivitas commit.

B. Analisis Optimasi Dockerfile

Sistem optimasi diberikan masukan berupa Dockerfile konvensional yang tidak beraturan, lalu dieksekusi menggunakan algoritma greedy berbasis DAG. Berikut adalah dockerfile asli sebelum dioptimasi:

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build
EXPOSE 3000
CMD ["npm", "start"]
```

Kemudian berikut ini Dockerfile hasil optimasi:

```
FROM node:18-alpine
WORKDIR /app
EXPOSE 3000
COPY package.json .
RUN npm install
COPY . .
RUN npm run build
CMD ["npm", "start"]
```

Hasil luaran dari sistem optimasi membuktikan bahwa algoritma bekerja sesuai dengan fungsi heuristik matematisnya. Terdapat dua pergerakan anomali positif yang dihasilkan oleh sistem:

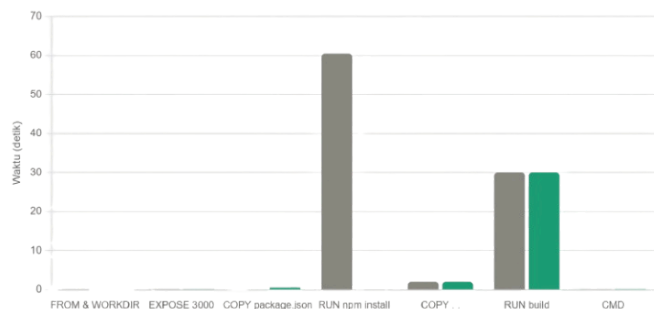
1. Pergeseran Simpul Statis: Instruksi EXPOSE 3000 awalnya berada di dasar berkas. Karena metrik volatilitas adalah 0.0 dan hanya bergantung pada WORKDIR, algoritma greedy menarik instruksi tersebut ke atas (baris ke-3). Hal ini mengunci layer tersebut secara permanen ke dalam cache/
2. Pemisahan instruksi COPY: Sistem secara otomatis memecah penyalinan berkas berdasarkan tingkat perubahannya. Berkas package.json yang jarang berubah diisolasi ke atas, menjadi prasyarat untuk RUN npm install. Sementara instruksi COPY . . yang membuat kode sumber bervolatilitas tinggi diturunkan tepat sebelum proses build. Ini adalah pembuktian empiris bahwa DAG berhasil menjaga logika kompilasi agar tidak mengalami fatal error meskipun urutannya dirombak total.

C. Evaluasi Kinerja

Eksperimen kompilasi ulang dilakukan dengan mensimulasikan perubahan pada source code. Pada Dockerfile asli, perubahan kode sumber memicu eksekusi instruksi COPY . . di baris ketiga, yang secara otomatis menginvalidasi cache untuk RUN npm install di baris keempat

TABLE I. PERBANDINGAN WAKTU KOMPILASI ULANG

Instruksi	Waktu Asli (s)	Waktu Teroptimasi (s)	Status Cache
FROM & WORKDIR	0.1	0.1	HIT
EXPOSE 3000	0.1	0.1	HIT
COPY package.json	-	0.5	HIT
RUN npm install	60.5	0.0	HIT
COPY .	2.0	2.0	MISS
RUN build	30.0	30.0	MISS
CMD	0.1	0.1	MISS
Total Waktu	92.8 (s)	32.8 (s)	



Berdasarkan data eksperimen, pendekatan algoritma greedy berhasil memangkas durasi kompilasi 92.8 detik menjadi hanya 32.8 detik. Penghematan waktu sebesar 64.6% ini didapatkan karena layer npm install berhasil diamankan ke dalam memori cache.

D. Pengaruh terhadap beban jaringan dan kapasitas registry

Selain metrik waktu, optimasi urutan graf instruksi ini berdampak langsung pada beban I/O jaringan CI/CD. Ketika cache pada instruksi npm install hilang pada arsitektur aslinya, server harus melakukan request eksternal ke repositori NPM publik untuk mengunduh ulang library yang ukurannya dapat mencapai ratusan megabyte.

Dalam skala industri, jika sebuah tim pengembang melakukan 20 kali commit dalam satu hari, Dockerfile yang tidak optimal akan memicu pemborosan bandwidth keluar-masuk sangat banyak per hari hanya untuk mengunduh library yang berulang. Algoritma greedy mereduksi anomali redundansi data ini mendekati nol, karena paket eksternal hanya akan diunduh jika nilai hash dari package.json benar-benar mengalami perubahan.

E. Limitasi Algoritma

Meskipun hasil simulasi menunjukkan optimasi yang signifikan, desain algoritma heuristik pada penelitian ini memiliki beberapa batasan yang perlu dipertimbangkan untuk implementasi produksi skala besar:

1. **Blindness Terhadap Runtime Dependencies:** Pendekatan DAG yang dibangun hanya bergantung pada analisis leksikal statis. Algoritma tidak dapat mendeteksi dependensi tidak tertulis. Misalnya, jika sebuah berkas konfigurasi database dibutuhkan pada saat npm run build, tetapi programmer tidak

menuliskannya secara eksplisit ke manifest graf, algoritma dapat melakukan pergeseran urutan yang berujung pada runtime error di dalam kontainer.

2. **Sub optimal pada arsitektur multi stage:** Makalah ini membatasi ruang lingkup pada Dockerfile tunggal bertipe single stage. Pada arsitektur kompilasi multi stage yang kompleks, algoritma greedy perlu dimodifikasi untuk melacak perpindahan antar-graf secara paralel, bukan sekedar penelusuran linier ke bawah.

V. IMPLEMENTASI DAN INTEGRASI PADA PIPELINE CI/CD

A. Arsitektur Integrasi Berbasis GitOps

Agar algoritma yang telah dirancang dapat berfungsi pada pengembangan perangkat lunak, program optimasi ini harus diintegrasikan ke dalam pipeline CI (Continuous Integration) menggunakan pendekatan GitOps.

Dalam arsitektur GitOps, repositori version control system bertindak sebagai kode sumber. Alur kerja integrasi sistem optimasi Dockerfile dirancang sebagai pre-build hook, dengan tahapan operasional sebagai berikut:

1. **Trigger / Webhook:** Developer melakukan push commit atau membuat pull request ke repositori utama
2. **Analysis Phase:** Server CI menjalankan runner yang akan mengunduh repositori. Kode python dieksekusi secara lokal di dalam agen tersebut untuk membaca riwayat .git dan mengekstrak volatilitas (V).
3. **Graph Execution:** Sistem mengonstruksi DAG, melakukan pencarian heuristik minimum, dan menimpa berkas Dockerfile lama dengan hasil optimasi.
4. **Build Phase:** Docker BuildKit mengambil alih eksekusi kompilasi menggunakan Dockerfile yang telah dioptimasi [4]

B. Implementasi Konfigurasi Pipeline (Github Actions)

Untuk mencoba integrasi, sistem optimasi ini dideklarasikan ke dalam konfigurasi YAML pada ekosistem Github Actions. Penggunaan konfigurasi ini memastikan bahwa optimasi berjalan otomatis tanpa ada aksi manual [5]. Berikut adalah contoh berkas konfigurasi Github Actions:

```
name: E-Ticketing CI Optimization
on:
  push:
    branches: [ "main", "staging" ]
jobs:
  build-and-optimize:
    runs-on: ubuntu-latest
    steps:
      # 1. Download source code(fetch-depth: 0)
      - name: Checkout Repository
        uses: actions/checkout@v3
        with:
          fetch-depth: 0

      # 2. Siapkan environment Python
      - name: Setup Python Environment
        uses: actions/setup-python@v4
        with:
```

```

python-version: '3.10'

# 3. Jalankan Algoritma Greedy untuk Optimasi
- name: Execute Greedy Dockerfile Optimizer
  run: |
    echo "Memulai analisis Volatilitas dan DAG..."
    python scripts/docker_optimizer.py --input
Dockerfile --output Dockerfile
    echo "Restrukturisasi selesai. Menampilkan
hasil:"
    cat Dockerfile

# 4. Kompilasi citra menggunakan Docker Buildx
- name: Build Docker Image
  uses: docker/build-push-action@v4
  with:
    context: .
    file: ./Dockerfile
    push: false
    tags: e-ticketing:latest
    cache-from: type=gha
    cache-to: type=gha,mode=max

```

Konfigurasi di atas adalah contoh penerapan pada CI yang sudah diterapkan ke project pribadi saya. Pada langkah ketiga konfigurasi di atas, kode secara otomatis menganalisis dan menimpa struktur. Pada langkah keempat, `cache-from: type=gha` dimanfaatkan secara penuh [4]. Karena instruksi berat seperti `RUN npm install` sudah ditarik ke atas oleh algoritma, fitur caching bawaan Github Actions (`gha`) dapat mengambil sisa komputasi sebelumnya secara utuh, mencegah server mengulangi proses instalasi yang berat

C. Penanganan kegagalan

Dalam penerapan algoritma heuristik pada lingkungan production, keandalan adalah faktor yang penting. Mengingat analisis DAG dilakukan secara statis, terdapat kemungkinan bahwa programmer menuliskan instruksi anti-pattern yang menghasilkan circular dependency, sehingga mengakibatkan sort gagal untuk menyelesaikannya.

Untuk mencegah terhentinya pipeline CI/CD akibat kegagalan optimasi sistem harus dilengkapi mekanisme fallback. Apabila fungsi eksekusi algoritma menemui blok eksepsi, kode tidak akan menghentikan proses, sebaliknya akan mencetak pesan peringatan ke dalam log server dan mengembalikan nilai kode dengan exit code 0, membiarkan Docker melakukan build menggunakan Dockerfile asli versi pemrogram. Keputusan desain ini memastikan bahwa integrasi optimasi ini bersifat fail-safe.

VI. KESIMPULAN DAN SARAN

A. Kesimpulan

Berdasarkan landasan teori, perancangan algoritma, dan hasil eksperimen yang telah diuraikan pada makalah ini, dapat ditarik beberapa kesimpulan utama:

1. Pemodelan urutan kompilasi Dockerfile ke dalam DAG merupakan prasyarat krusial yang memungkinkan algoritma melakukan pemindahan baris instruksi tanpa merusak ketergantungan dalam pengembangan perangkat lunak.
2. Penerapan algoritma greedy yang didukung heuristik indeks volatilitas (V) secara konsisten berhasil memisahkan lapisan statis dari lapisan dinamis.

3. Hasil evaluasi empiris menunjukkan bahwa Dockerfile hasil optimasi algoritma greedy mampu mencegah invalidasi pada layer. Pada skenario kompilasi ulang, waktu eksekusi server berhasil dikurangi secara signifikan dari 92.8 detik menjadi 32.8 detik, menghasilkan tingkat efisiensi waktu komputasi sebesar 64.6% tanpa membebani memori.

B. Saran

Meskipun hasil simulasi membuktikan keberhasilan optimasi, terdapat beberapa aspek yang dapat dikembangkan selanjutnya:

1. Analisis multi stage build: Untuk selanjutnya disarankan untuk memodifikasi algoritma agar mampu menangani Dockerfile berarsitektur multi stage, di mana DAG harus mampu mendeteksi perpindahan antar fase kompilasi.
2. Penyesuaian bobot harus didasarkan pada semua riwayat commit.

PRANALA REPOSITORI

<https://github.com/fazridep/dockerfile-optimize>

REFERENSI

- [1] Docker Inc., "Building Best Practices," Docker Documentation. [Online]. Available: <https://docs.docker.com/build/building/best-practices/>. [Accessed: Jun. 17, 2026].
- [2] Docker Inc., "Build Cache," Docker Documentation. [Online]. Available: <https://docs.docker.com/build/cache/>. [Accessed: Jun. 17, 2026].
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.
- [4] Docker Inc., "BuildKit Overview," Docker Documentation. [Online]. Available: <https://docs.docker.com/build/buildkit/>. [Accessed: Jun. 17, 2026].
- [5] GitHub Inc., "GitHub Actions Documentation." [Online]. Available: <https://docs.github.com/actions> [Accessed: Jun. 17, 2026].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 18 Juni 2026



Fazri Arrashy Putra