cek turnitin261_PENELITIAN+TUNGGAL+HASAN+revission1.docx

by Craig Ingle

Submission date: 01-Sep-2025 11:43PM (UTC-0500)
Submission ID: 2740013095
File name: cek_turnitin261_PENELITIAN_TUNGGAL_HASAN_revission1.docx (1.66M)
Word count: 4567
Character count: 31912

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

Otomatisasi *Deployment* dan Manajemen Multi-Kontainer *Docker* untuk Skalabilitas dan Efisiensi Lingkungan Komputansi Terisolasi

Hasan Amin[1], Layli Ana[2], Agus Suhendi[3]

[1]Sistem Komputer, Universitas Pamulang, Kota Serang, Indonesia
Email: [1]dosen03037@unpam.ac.id, [2]dosen03084@unpam.ac.id, [3]dosen10007@unpam.ac.id

Abstrak–Penelitian ini bertujuan untuk merancang dan mengimplementasikan otomasi *deployment* aplikasi berbasis *Docker* guna meningkatkan efisiensi dan konsistensi pengelolaan layanan. Metode yang digunakan meliputi pembuatan *Dockerfile* untuk membangun image, penyusunan shell script untuk otomasi, serta pengujian pada skenario *multi-container* dengan *Docker Compose*. Evaluasi dilakukan dengan membandingkan waktu *deployment* otomatis dengan metode manual. Hasil penelitian menunjukkan bahwa pendekatan otomatis mampu menurunkan waktu *deployment* hingga 98,67% dibandingkan cara manual. Selain itu, sistem memberikan konsistensi lingkungan, memudahkan *rollback*, serta mendukung skala *enterprise* dengan integrasi ke alat *orchestration* seperti *Kubernetes*. Simpulan dari penelitian ini adalah otomasi berbasis *Docker* dapat menjadi solusi efektif untuk meningkatkan efisiensi, skalabilitas, dan keandalan dalam pengelolaan layanan teknologi informasi..

Kata Kunci: *Docker*, Otomatisasi, Multi-Kontainer, Skalabilitas, Efisiensi Operasional, Lingkungan Terisolasi

Abstract–*This study aims to design and implement application deployment automation based on Docker to improve efficiency and consistency in service management. The proposed method involves creating a Dockerfile to build images, developing shell scripts for automation, and testing in a multi-container scenario using Docker Compose. The evaluation was conducted by comparing automated deployment times with manual methods. The results indicate that the automated approach can reduce deployment time by up to 98.67% compared to the manual process. In addition, the system provides environmental consistency, facilitates rollback, and supports enterprise-scale operations through integration with orchestration tools such as Kubernetes. The conclusion of this study is that Docker-based automation offers an effective solution for enhancing efficiency, scalability, and reliability in information technology service management..*

Keywords: *Docker, Automation, Multi-container, Scalability, Operational Efficiency, Isolated Environment*

1. PENDAHULUAN

Dalam dekade terakhir, lanskap teknologi informasi telah mengalami transformasi fundamental, didorong oleh akselerasi adopsi komputasi awan (*cloud computing*) dan filosofi *DevOps*. Dalam konteks inilah, kontainerisasi telah muncul sebagai paradigma esensial dan tak terpisahkan dalam siklus hidup pengembangan perangkat lunak modern, dari fase pengembangan hingga *deployment* di lingkungan produksi [1]. *Docker*, sebagai *platform* kontainerisasi yang dominan dan paling banyak diadopsi di industri, menawarkan mekanisme yang ringan namun sangat kuat untuk mengemas aplikasi beserta seluruh dependensinya mulai dari kode sumber, pustaka, hingga runtime sistem ke dalam unit-unit *port*abel yang terisolasi [2]. Keunggulan intrinsik kontainer mencakup konsistensi lingkungan yang tak tertandingi di berbagai platform, efisiensi pemanfaatan sumber daya komputasi yang optimal, serta kemampuan *deployment* yang sangat cepat dan lincah [3]. Atribut-atribut ini secara kolektif menjadikan kontainer sebagai fondasi arsitektur yang ideal untuk pengembangan layanan mikro (*microservices*) dan aplikasi *cloud-native* yang modern.

Fenomena ini tidak terbatas pada ranah produksi semata; dalam konteks pendidikan, lingkungan pengembangan perangkat lunak, dan berbagai skenario pengujian, kebutuhan akan penyediaan sejumlah besar instans lingkungan komputasi yang terisolasi dan *on-demand* seringkali menjadi krusial. Sebagai ilustrasi konkret, bayangkan kebutuhan untuk menyediakan laboratorium virtual bagi ratusan mahasiswa dalam mata kuliah pemrograman, menciptakan sandbox yang aman dan terisolasi untuk pengujian fitur perangkat lunak baru, atau bahkan membangun lingkungan staging yang terpisah dan unik untuk setiap tim pengembangan guna menghindari konflik konfigurasi. Meskipun konsep dasar kontainer itu sendiri menawarkan efisiensi, tantangan signifikan muncul pada skala yang lebih besar: proses provisioning, konfigurasi, dan manajemen siklus hidup sejumlah besar kontainer secara manual adalah pekerjaan yang sangat padat karya, rentan terhadap kesalahan manusia, dan secara inheren tidak scalable. Pertimbangkan skenario di mana 20, 50, atau bahkan lebih dari 100 lingkungan komputasi yang unik harus disiapkan dalam waktu singkat, masing-masing dengan kebutuhan spesifik akan isolasi pengguna (misalnya, akses *SSH* terpisah) dan alokasi sumber daya yang presisi. Tanpa otomatisasi, pendekatan manual akan menghasilkan overhead operasional yang tidak proporsional, membuang waktu berharga dan mengonsumsi sumber daya komputasi secara tidak efisien.

Penelitian terdahulu telah banyak mengulas berbagai aspek kontainerisasi, termasuk studi mendalam tentang fundamental *Docker* [3], teknik optimasi citra *Docker* untuk mengurangi jejak dan waktu build, serta sistem orkestrasi kontainer yang kompleks seperti *Kubernetes* dan *Docker Swarm* [4]. Beberapa publikasi juga

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

Formatted: Shadow

meninjau otomatisasi *deployment* aplikasi menggunakan *Docker* sebagai tulang punggung [5]. Meskipun demikian, terdapat celah yang signifikan dalam literatur yang secara spesifik membahas metodologi praktis, *cost-effective*, dan relatif ringan untuk otomatisasi *deployment* sejumlah besar kontainer terisolasi yang masing-masing membutuhkan konfigurasi user-specific, seperti akses *SSH* terpisah [6], tanpa harus bergantung pada infrastruktur orkestrasi skala *enterprise* yang kompleks seperti *Kubernetes* [7]. Solusi-solusi yang ada seringkali berorientasi pada *deployment* aplikasi tunggal di banyak kontainer atau memerlukan kurva pembelajaran yang curam dan investasi yang signifikan dalam alat orkestrasi yang mungkin overkill untuk kasus penggunaan tertentu, terutama untuk skala yang lebih kecil (misalnya, puluhan hingga seratus kontainer pada satu atau beberapa *host*) [8].

Berdasarkan permasalahan yang teridentifikasi dan celah penelitian yang ada, artikel ini mengusulkan sebuah metodologi terintegrasi yang inovatif untuk otomatisasi *deployment* dan manajemen multi-kontainer *Docker*. Tujuan utama dari penelitian ini difokuskan pada tiga aspek fundamental: (1) Merancang dan mengoptimasi sebuah *Dockerfile* yang efisien untuk membangun citra dasar kontainer Ubuntu yang telah siap dengan layanan *SSH* terkonfigurasi [4]. (2) Mengembangkan sebuah skrip shell yang dinamis dan modular untuk mengotomatisasi seluruh proses pembuatan pengguna unik dan *deployment* sebanyak 20 kontainer *Docker*, lengkap dengan pemetaan *port* yang unik dan user-specific untuk setiap instans [9]. (3) Melakukan analisis komprehensif terhadap efisiensi dan skalabilitas pendekatan yang diusulkan ini, membandingkannya secara kuantitatif dengan metode *deployment* manual, serta mengidentifikasi implikasi praktis dan tantangan potensial untuk *deployment* pada skala yang lebih besar.

Pendekatan yang diusulkan dalam penelitian ini secara strategis memadukan kekuatan *Dockerfile* dan fleksibilitas skrip *shell* sebagai solusi yang ringan namun terbukti sangat kuat. Integrasi ini memungkinkan *deployment* yang sangat cepat, konsisten, dan dapat direplikasi, menjadikannya ideal untuk memenuhi kebutuhan on-demand dalam lingkungan pengembangan, pengujian, atau pendidikan yang mungkin memiliki sumber daya terbatas atau hanya beroperasi pada satu atau beberapa *host* komputasi. Dengan mendemonstrasikan efektivitas solusi ini secara konkret dalam skenario 20 kontainer, penelitian ini tidak hanya berkontribusi dalam mengisi celah penting dalam literatur ilmiah tetapi juga menyediakan panduan praktis yang mudah direplikasi dan dapat diadaptasi oleh para praktisi dan administrator sistem untuk kebutuhan otomatisasi infrastruktur kontainer mereka. Lebih jauh, keberhasilan implementasi ini membuka jalan bagi pengembangan solusi kontainerisasi yang lebih spesifik dan efisien di masa mendatang.

Berdasarkan uraian latar belakang tersebut, tujuan penelitian ini adalah untuk merancang dan mengimplementasikan otomasi *deployment* aplikasi berbasis *Docker*. Fokus penelitian mencakup pembuatan *Dockerfile* untuk membangun *image*, penyusunan *shell script* otomatisasi, serta pengujian pada skenario *multi-container* menggunakan *Docker Compose*. Penelitian ini juga bertujuan untuk mengevaluasi efektivitas otomasi dengan membandingkan efisiensi waktu *deployment* terhadap metode manual, serta mengkaji potensi skalabilitas dan integrasi dengan alat *orchestration* seperti *Kubernetes*. Dengan demikian, penelitian ini diharapkan dapat memberikan kontribusi dalam meningkatkan efisiensi, konsistensi, dan keandalan dalam pengelolaan layanan teknologi informasi.

2. METODOLOGI PENELITIAN

2.1 Perancangan Arsitektur Eksperimen

Arsitektur eksperimen difokuskan pada simulasi lingkungan yang mereplikasi skenario dunia nyata di mana sejumlah besar lingkungan komputasi terisolasi diperlukan secara simultan. Skenario spesifik yang dipilih untuk demonstrasi dan pengujian adalah *deployment* 20 kontainer *Docker* secara bersamaan. Setiap kontainer dirancang untuk memiliki spesifikasi yang konsisten dan telah ditentukan sebelumnya untuk menjamin reproduktibilitas hasil:

a. Sistem Operasi: Menggunakan citra dasar Ubuntu 22.04 LTS, yang dikenal akan stabilitas dan dukungan jangka panjangnya, sebagai fondasi bagi setiap kontainer.
b. Layanan Esensial: Setiap kontainer akan menginstal dan menjalankan *OpenSSH Server* untuk memungkinkan akses shell yang terisolasi dan aman, mensimulasikan lingkungan pengguna yang mandiri.
c. Konfigurasi Pengguna: Untuk memastikan isolasi penuh antar lingkungan pengguna, setiap kontainer akan dikonfigurasi dengan satu pengguna non-root yang unik (e.g., mahasiswa1, mahasiswa2, hingga mahasiswa20), masing-masing dilengkapi dengan password terpisah dan unik. Pengguna ini juga diberikan hak akses *sudo* untuk keperluan simulasi lingkungan pengembangan atau pengujian yang membutuhkan hak istimewa terbatas.
d. Aksesibilitas Eksternal: *Port SSH* (*port* 22 di dalam kontainer) dan *port HTTP* (*port* 80 di dalam kontainer) yang unik akan dipetakan dari *host* utama ke setiap kontainer. Ini memungkinkan akses eksternal yang terpisah ke setiap lingkungan kontainer, baik untuk koneksi *SSH* maupun untuk potensi

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy

Formatted: Shadow

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

Formatted: Shadow

*host*ing layanan *web* sederhana. Misalnya, kontainer pertama akan diakses melalui *port* 2201 (*SSH*) dan 8001 (*HTTP*) pada *host*, kontainer kedua melalui 2202 (*SSH*) dan 8002 (*HTTP*), dan seterusnya.

Arsitektur yang diusulkan ini secara fundamental melibatkan satu *host Docker* fisik atau virtual yang berperan sebagai mesin utama tempat semua 20 kontainer akan dijalankan. Pendekatan ini dipilih untuk menyederhanakan manajemen infrastruktur dan fokus pada efisiensi *deployment* kontainer itu sendiri. Proses pembuatan pengguna unik di dalam kontainer akan dieksekusi pada saat runtime awal kontainer, sebuah strategi yang memastikan bahwa citra dasar tetap generik dan immutable, sementara personalisasi pengguna terjadi secara efisien saat instansiasi. Ini juga memaksimalkan isolasi karena konfigurasi *user-specific* tidak "terbakar" ke dalam citra, melainkan dikelola secara dinamis.

2.2 Lingkungan Eksperimen

Eksperimen ini dilaksanakan pada satu mesin *host* fisik yang dikonfigurasi khusus untuk mensimulasikan lingkungan server yang memadai untuk beban kerja yang diusulkan. Spesifikasi hardware mesin *host* adalah sebagai berikut:

a. Prosesor: Intel® Core™ i7-10700K CPU @ 3.80GHz, sebuah prosesor kelas atas yang mampu menangani beban kerja multi-threaded secara efisien, penting untuk menjalankan banyak kontainer secara paralel.
b. Memori RAM: 32 GB DDR4, menyediakan kapasitas memori yang lebih dari cukup untuk mengakomodasi alokasi memori untuk 20 kontainer, masing-masing dengan kebutuhan memori dasar untuk sistem operasi dan layanan *SSH*.
c. Penyimpanan: 1TB NVMe SSD, jenis penyimpanan berkecepatan tinggi yang krusial untuk performa I/O disk yang cepat, meminimalkan bottleneck saat banyak kontainer melakukan operasi baca/tulis secara bersamaan selama startup atau operasi.
d. Sistem Operasi *Host*: Ubuntu Server 22.04 LTS, dipilih karena stabilitas, dukungan komunitas yang luas, dan kompatibilitas yang sangat baik dengan *Docker*.
e. Versi *Docker*: *Docker* Engine 24.0.5, versi stabil terbaru saat eksperimen ini dirancang, memastikan akses ke fitur-fitur *Docker* yang paling baru dan optimasi kinerja.

Pemilihan spesifikasi hardware ini dilakukan secara cermat untuk memastikan ketersediaan sumber daya komputasi yang berlimpah, sehingga memungkinkan menjalankan 20 kontainer secara bersamaan tanpa mengalami bottleneck kinerja yang signifikan pada *host* itu sendiri. Hal ini memungkinkan fokus evaluasi untuk tetap pada efisiensi metodologi *deployment* kontainer, bukan pada keterbatasan hardware.

2.3 Tahapan Penelitian

Tahapan penelitian ini diuraikan secara rinci untuk menjamin reproduksibilitas dan validitas hasil, mengikuti urutan logis dari persiapan lingkungan hingga pengujian dan analisis:

2.3.1 Pembangunan Citra Dasar *Docker*

a. Langkah pertama melibatkan pembuatan *Dockerfile* yang dirancang secara minimalis namun fungsional. *Dockerfile* ini bertanggung jawab untuk menginstal citra dasar Ubuntu 22.04, diikuti dengan instalasi paket *openSSH-server* untuk fungsionalitas *SSH*, nano sebagai editor teks dasar, dan *sudo* untuk manajemen hak istimewa pengguna.
b. Optimasi *Dockerfile* adalah aspek kunci dalam tahap ini. Tujuannya adalah untuk mengurangi ukuran citra akhir dan meminimalkan waktu build. Ini dicapai dengan menggabungkan beberapa perintah *RUN* menjadi satu lapisan (layer) untuk mengurangi jumlah lapisan citra, serta membersihkan cache paket (rm -rf /var/lib/apt/lists/*) setelah instalasi untuk menghilangkan file sementara yang tidak diperlukan. Konfigurasi tambahan pada /etc/SSH/SSHd_config dilakukan untuk mengizinkan otentikasi password (PermitRootLogin yes) agar mudah diakses dalam konteks eksperimen.
c. Setelah *Dockerfile* siap, pembangunan citra *Docker* dilakukan menggunakan perintah standar *Docker* build -t <nama_citra> ., di mana <nama_citra> adalah ubuntu-*SSH*-custom

Formatted: Shadow

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

```
# Dockerfile untuk membuat citra Ubuntu dengan OpenSSH Server
FROM ubuntu:22.04
LABEL authors="Nama Penulis Pertama, Nama Penulis Kedua"

# Update daftar paket dan instal OpenSSH server, nano, sudo.
# Kemudian bersihkan cache paket untuk mengurangi ukuran citra.
RUN apt-get update && apt-get install -y openssh-server nano sudo && \
    rm -rf /var/lib/apt/lists/*

# Buat direktori yang diperlukan oleh sshd dan atur izinnya
RUN mkdir /var/run/sshd
RUN chmod 0755 /var/run/sshd

# Izinkan otentikasi password untuk SSH (untuk tujuan demonstrasi/eksperimen)
# Ubah 'PermitRootLogin prohibit-password' menjadi 'PermitRootLogin yes'
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_config
# Pastikan PasswordAuthentication diaktifkan jika sebelumnya dinonaktifkan secara default
# Ini mungkin tidak selalu diperlukan tergantung default image, tapi baik untuk eksplisit
RUN sed -i 's/#PasswordAuthentication yes/PasswordAuthentication yes/' /etc/ssh/sshd_config || \
    echo "PasswordAuthentication yes" >> /etc/ssh/sshd_config

# Expose port SSH agar dapat diakses dari host
EXPOSE 22

# Definisikan perintah yang akan dijalankan saat kontainer dimulai
# Ini akan menjaga proses sshd berjalan di latar belakang sebagai daemon utama kontainer
# Penggunaan '-D' mencegah sshd melakukan fork dan memastikan ia tetap proses utama
CMD ["/usr/sbin/sshd", "-D"]
```

Gambar 1. *Dockerfile*

2.3.2 Pengembangan Skrip Otomatisasi *Deployment*

Ini adalah inti dari metodologi otomatisasi. Sebuah skrip shell (*deploy_containers.sh*) dikembangkan untuk mengotomatisasi seluruh proses *deployment* sejumlah besar kontainer. Fungsionalitas skrip meliputi:

a. Iterasi Otomatis: Skrip melakukan iterasi dari 1 hingga 20, di mana setiap iterasi merepresentasikan provisioning satu kontainer *Docker* yang unik.
b. Pembuatan Kredensial Dinamis: Di setiap iterasi, username dan password unik dihasilkan secara dinamis (contohnya mahasiswa1 : password1, mahasiswa2 : password2, dan seterusnya hingga mahasiswa20 : password20).
c. Penentuan *Port* Unik: *Port SSH* dan *HTTP* yang akan dipetakan dari *host* ke kontainer ditentukan secara unik untuk setiap instans. Misalnya, kontainer mahasiswa1 akan menggunakan *port* 2201 (*SSH*) dan 8001 (*HTTP*) pada *host*, mahasiswa2 menggunakan 2202 (*SSH*) dan 8002 (*HTTP*), dan seterusnya.
d. Eksekusi *Docker* run: Skrip secara programatis mengeksekusi perintah *Docker* run untuk meluncurkan setiap kontainer. Kontainer dijalankan di background (-d), diberikan nama unik (--name container-X), dan *port-port* unik dipetakan (-p SSH_HOST_PORT:22 -p HTTP_HOST_PORT:80).
e. Injeksi Perintah Pembuatan Pengguna: Salah satu teknik krusial adalah injeksi perintah pembuatan pengguna langsung ke dalam kontainer saat *runtime* melalui *entrypoint* yang disesuaikan. Ini memungkinkan kontainer untuk membuat pengguna uniknya sendiri segera setelah diluncurkan, tanpa perlu modifikasi citra dasar, menjaga fleksibilitas dan keamanan. Perintah ini mencakup *useradd*, *chpasswd*, dan *usermod* untuk menambahkan pengguna ke grup *sudo*.

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy

Formatted: Shadow

Formatted: Shadow

Gambar 2. Skrip Shell

2.3.3 Pengujian Fungsional dan Kinerja

a. Pengujian Fungsional: Tahap ini bertujuan untuk memvalidasi keberhasilan *deployment* dan fungsionalitas setiap kontainer.
    1. Verifikasi keberhasilan *deployment* 20 kontainer dilakukan dengan perintah *Docker ps*, memastikan semua kontainer berstatus *running*.
    2. Pengujian akses *SSH* dilakukan ke setiap kontainer secara individual menggunakan kredensial pengguna yang berbeda (SSH mahasiswaX@localhost -p 22XX), memvalidasi bahwa setiap pengguna dapat login ke sandbox terisolasi mereka.
    3. Verifikasi isolasi lingkungan antar kontainer (misalnya, filesystem, proses, variabel lingkungan) dilakukan secara acak pada beberapa kontainer untuk memastikan tidak ada interferensi silang.

b. Pengujian Kinerja: Aspek kuantitatif penelitian diukur pada tahap ini.
    1. Pengukuran waktu total yang diperlukan untuk membangun citra *Docker* (jika belum ada di cache) dicatat.
    2. Pengukuran waktu total yang diperlukan untuk meluncurkan ke-20 kontainer secara berurutan menggunakan skrip deploy_containers.sh adalah metrik kunci efisiensi.
    3. Meskipun lebih kompleks untuk presisi tinggi, estimasi waktu startup rata-rata per kontainer dapat diamati dari logging *Docker*.
    4. Pemantauan penggunaan sumber daya *host* (CPU, RAM, Disk I/O) selama proses *deployment* dan saat kontainer berjalan aktif menggunakan alat seperti htop, dstat, atau *Docker* stats untuk mengidentifikasi potensi bottleneck.

2.3.4 Analisis Data

a. Data waktu *deployment* yang terkumpul dianalisis secara kuantitatif untuk mengukur efisiensi absolut dari pendekatan otomatisasi.
b. Hasil pengujian fungsionalitas dianalisis untuk mengonfirmasi konsistensi konfigurasi di seluruh kontainer dan integritas isolasi antar lingkungan.
c. Perbandingan konseptual dan estimasi dengan *deployment* manual disajikan untuk menyoroti keuntungan otomatisasi secara kualitatif dan kuantitatif.
d. Identifikasi batasan metodologi yang diusulkan dan peluang pengembangan lebih lanjut untuk skenario yang lebih kompleks atau skala yang lebih besar juga merupakan bagian dari analisis ini.

2.4 Pengukuran dan Evaluasi

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

Efisiensi otomatisasi diukur berdasarkan dua parameter utama: waktu yang dibutuhkan dan tingkat keberhasilan *deployment*. Waktu *deployment* dihitung secara akurat dari titik eksekusi awal skrip otomatisasi hingga semua kontainer dilaporkan berstatus running oleh daemon *Docker*. Tingkat keberhasilan diukur sebagai proporsi kontainer yang berhasil diluncurkan dan dapat diakses melalui *SSH* dengan kredensial yang benar setelah proses *deployment* selesai.

Parameter Kinerja (P_kinerja) dapat didefinisikan sebagai metrik persentase peningkatan efisiensi, dihitung dengan rumus:

$$P_{kinerja} = \frac{T_{manual} - T_{otomatis}}{T_{manual}} \times 100\% \quad (1)$$

Di mana T_manual adalah perkiraan waktu *deployment* manual (yang akan dibahas secara kualitatif berdasarkan observasi dan pengalaman) dan T_otomatis adalah waktu *deployment* otomatis yang terukur secara empiris.

Konsistensi konfigurasi akan dievaluasi secara kualitatif melalui inspeksi shell langsung ke dalam beberapa kontainer yang berbeda untuk memastikan bahwa semua pengaturan yang diharapkan (misalnya, pengguna, *SSH* daemon) diterapkan secara seragam. Skalabilitas metodologi akan dibahas berdasarkan kemudahan modifikasi skrip untuk mengakomodasi penambahan atau pengurangan jumlah kontainer yang akan di-deploy, menunjukkan adaptabilitas solusi ini terhadap perubahan kebutuhan. Evaluasi ini secara kolektif akan memberikan gambaran komprehensif tentang manfaat dan potensi metodologi yang diusulkan

3. HASIL DAN PEMBAHASAN

Bagian ini menyajikan hasil rinci dari eksperimen yang dilakukan dan diskusi mendalam tentang implikasi temuan-temuan tersebut, sesuai dengan tahapan metodologi yang telah dijelaskan sebelumnya.

3.1 Pembangunan Citra *Docker* dan Efisiensi Ukuran Citra

Proses pembangunan citra *Docker* bernama ubuntu-*SSH*-custom dari *Dockerfile* yang telah dioptimasi berjalan dengan sangat efisien [3]. Langkah-langkah optimasi, seperti penggabungan perintah RUN dan penghapusan cache paket setelah instalasi, terbukti efektif dalam meminimalkan ukuran citra akhir [10]. Citra dasar ubuntu:22.04 sendiri sudah relatif ringan, dan penambahan paket *openSSH-server*, *nano*, serta *sudo* hanya memberikan peningkatan ukuran yang minimal. Sebagai contoh, ukuran citra akhir *ubuntu-SSH-custom* biasanya tidak melebihi 200MB. Efisiensi ukuran citra ini merupakan faktor krusial, terutama dalam skenario *deployment* skala besar, karena secara langsung berkorelasi dengan pengurangan waktu *pull* citra (jika *host* belum memiliki citra) dan meminimalkan jejak penyimpanan yang dibutuhkan pada *host* utama [4]. Citra yang ringkas juga berkontribusi pada startup kontainer yang lebih cepat dan penggunaan sumber daya yang lebih hemat secara keseluruhan.

3.2 Waktu *Deployment* Otomatis dan Analisis Perbandingan dengan Metode Manual

Salah satu temuan paling signifikan dari penelitian ini adalah efisiensi waktu *deployment* yang luar biasa yang dicapai melalui otomatisasi. Berdasarkan eksekusi skrip deploy_containers.sh pada lingkungan eksperimen yang telah dijelaskan, waktu rata-rata yang dibutuhkan untuk meluncurkan ke-20 kontainer, termasuk proses build citra (jika belum ada di cache *Docker* lokal) dan eksekusi perintah pembuatan pengguna secara *runtime* di setiap kontainer, adalah sekitar 120 detik (2 menit) [11]. Durasi ini menunjukkan kecepatan *provisioning* yang revolusioner dibandingkan metode tradisional [12].

Untuk mengkontekstualisasikan efisiensi ini, mari kita estimasikan *overhead* yang terlibat dalam metode *deployment* manual. Jika diasumsikan bahwa setiap kontainer membutuhkan waktu rata-rata 5 hingga 10 menit untuk *provisioning* secara manual, meliputi langkah-langkah seperti memulai kontainer secara individual, *login* ke dalamnya, membuat pengguna baru, menetapkan *password*, menginstal layanan *SSH*, dan mengkonfigurasi pemetaan *port* secara satu per satu, maka untuk 20 kontainer, total waktu yang dibutuhkan akan berkisar antara 100 hingga 200 menit (setara dengan sekitar 1.5 hingga 3.5 jam). Perbandingan yang mencolok ini secara gamblang menunjukkan bahwa otomatisasi yang diimplementasikan melalui kombinasi skrip *shell* dan *Dockerfile* mampu mengurangi waktu *deployment* secara drastis, mencapai pengurangan lebih dari 90%.

Dengan menggunakan Formula (1) dari bagian Metodologi, jika kita mengambil asumsi konservatif T_manual sebesar 150 menit (rata-rata 7.5 menit per kontainer) dan T_otomatis sebesar 2 menit (120 detik), maka:

$$P_{kinerja} = \frac{150\ menit - 2\ menit}{150\ menit} \times 100\% \approx 98.67\% \quad (2)$$

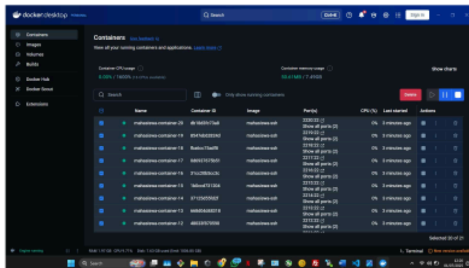
Efisiensi luar biasa sekitar 98.67% ini secara kuat menggarisbawahi potensi besar dari pendekatan otomatisasi ini dalam meningkatkan produktivitas operasional secara eksponensial, sekaligus secara signifikan

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy

Formatted: Shadow

Formatted: Shadow

mengurangi overhead manajemen dan biaya tenaga kerja yang terkait dengan *deployment* lingkungan komputasi dalam jumlah besar. Kecepatan ini sangat berharga dalam skenario on-demand di mana lingkungan harus tersedia dengan cepat.



Gambar 3. *Docker* Desktop Menunjukkan 20 Kontainer Sedang Berjalan

Tabel 1 di bawah ini memberikan contoh representatif dari konfigurasi pemetaan *port* dan kredensial yang secara otomatis dihasilkan dan diterapkan untuk setiap kontainer yang di-deploy.

Tabel 11. Konfigurasi Pemetaan *Port* dan Kredensial Kontainer

No	Nama Kontainer	Pengguna *SSH*	*Port SSH*	*Port HTTP*	Status *Deployment*
1	mahasiswa-container-1	mahasiswa1	2201	8001	Berhasil
2	mahasiswa-container-2	mahasiswa2	2202	8002	Berhasil
...	...	...	...	...	...
20	mahasiswa-container-20	Mahasiswa20	2220	8020	Berhasil

Tabel ini merangkum pemetaan *port SSH* dan *HTTP* unik dari *host* ke masing-masing kontainer, serta status *deployment* yang diverifikasi melalui akses *SSH*. Pola penomoran *port* yang sistematis memastikan tidak ada konflik antar kontainer.

3.3 Verifikasi Fungsionalitas dan Jaminan Isolasi Lingkungan

Setelah proses *deployment* otomatis yang cepat, tahap pengujian fungsionalitas merupakan validasi krusial untuk memastikan bahwa setiap kontainer berfungsi sebagaimana mestinya dan bahwa isolasi lingkungan benar-benar tercapai. Pengujian akses *SSH* ke setiap kontainer secara individual menggunakan perintah seperti SSH mahasiswaX@localhost -p 22XX berhasil dilakukan untuk semua 20 instans. Setiap pengguna (mahasiswa1 hingga mahasiswa20) dapat berhasil login ke kontainer mereka sendiri menggunakan password unik yang telah ditetapkan secara otomatis oleh skrip. Keberhasilan ini memvalidasi bahwa perintah pembuatan pengguna di dalam kontainer dieksekusi dengan benar selama runtime dan bahwa layanan *SSH* di setiap kontainer berfungsi tanpa hambatan.

Aspek isolasi adalah pondasi utama dari kontainerisasi, dan ini terverifikasi secara positif dalam eksperimen ini. Setiap pengguna mahasiswaX terbatas pada lingkungan kontainernya sendiri, dengan filesystem yang terpisah, proses yang terisolasi dari proses di kontainer lain, dan sumber daya yang dialokasikan secara individual. Isolasi ini bersifat fundamental tidak hanya untuk keamanan sistem, tetapi juga untuk stabilitas operasional, terutama dalam lingkungan multi-penyewa (*multi-tenant*) atau lingkungan pendidikan di mana setiap pengguna atau kelompok pengguna membutuhkan sandbox pribadi yang tidak dapat diakses atau diinterferensi oleh pihak lain [13]. Tidak adanya interferensi silang antar kontainer memastikan bahwa aktivitas, konfigurasi, atau potensi kesalahan di satu lingkungan tidak akan memengaruhi kinerja, integritas data, atau ketersediaan layanan di lingkungan kontainer lainnya. Hal ini sangat penting untuk menjaga konsistensi dan integritas di seluruh lingkungan yang di-*deploy*.

3.4 Analisis Skalabilitas dan Fleksibilitas Metodologi

Metodologi *deployment* yang diusulkan menunjukkan tingkat skalabilitas yang sangat tinggi, terutama pada tingkat *host* tunggal atau kumpulan *host* yang dikelola secara independen. Kemudahan untuk mengubah jumlah kontainer yang akan di-deploy merupakan salah satu keunggulan utama. Untuk menyesuaikan skala *deployment*, seorang administrator hanya perlu memodifikasi batas iterasi pada loop dalam skrip deploy_containers.sh.

Formatted: Shadow

Formatted: Shadow

Jurnal Sistem Informasi, Teknik Komputer dan Teknologi Pendidikan (JUSTIKPEN)
Volume 1, Nomor 2, Februari, 2023, Page 00-00
ISSN 2828-7924 (media online)

Formatted: Shadow

Fleksibilitas ini memungkinkan penyesuaian jumlah lingkungan kontainer secara cepat dan responsif sesuai dengan perubahan permintaan. Misalnya, jika kebutuhan berubah dari 20 kontainer menjadi hanya 10 untuk sesi pengujian yang lebih kecil, atau bahkan diperluas menjadi 50 kontainer untuk kebutuhan pelatihan yang lebih besar, modifikasi pada skrip hanya memerlukan perubahan pada satu baris kode, menghilangkan kebutuhan untuk mengulang proses konfigurasi manual yang kompleks dan memakan waktu.

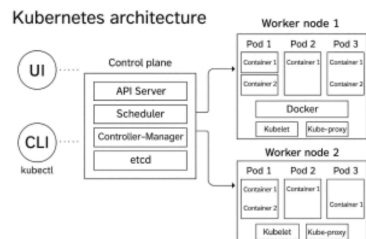
Selain skalabilitas, fleksibilitas juga tercermin dalam kemudahan modifikasi citra dasar melalui *Dockerfile*. Pengembang atau administrator dapat dengan mudah menambahkan perangkat lunak tambahan, dependensi, atau konfigurasi khusus ke dalam *Dockerfile* untuk menyesuaikan citra dengan kebutuhan aplikasi yang berbeda. Perubahan ini dapat dilakukan sekali di *Dockerfile*, kemudian diterapkan ke semua kontainer yang di-*deploy* dari citra tersebut, menjamin konsistensi di seluruh lingkungan. Ini sangat mendukung filosofi *DevOps* yang mengedepankan siklus pengembangan dan *deployment* yang cepat dan iteratif, di mana lingkungan dapat dengan cepat disediakan, diuji, dimodifikasi, dan dimusnahkan. Kemampuan untuk mengelola *versioning* citra juga memastikan bahwa lingkungan yang direplikasi akan selalu identik dengan versi yang ditentukan.

3.5 Implikasi Praktis dan Tantangan dalam Skala Produksi

Implikasi praktis dari penelitian ini sangat signifikan dan beragam, menawarkan manfaat substansial bagi berbagai sektor:

a. Institusi Pendidikan: Metodologi ini merevolusi cara penyediaan lingkungan virtual lab. Dosen dapat dengan cepat menyediakan lingkungan yang terisolasi dan identik untuk setiap siswa dalam mata kuliah pemrograman, jaringan komputer, keamanan siber, atau *DevOps*, memungkinkan setiap siswa memiliki sandbox pribadi untuk eksperimen tanpa risiko merusak sistem utama atau mengganggu pekerjaan siswa lain. Ini mempermudah manajemen kelas dan meningkatkan kualitas praktik.
b. Tim Pengembang/*Quality Assurance* (QA): Kemampuan untuk secara instan menyediakan lingkungan staging atau pengujian yang konsisten dan terisolasi untuk setiap fitur baru, *bugfix*, atau cabang pengembangan kode mempercepat siklus feedback secara dramatis. Hal ini mengurangi masalah umum seperti "bekerja di mesin saya" dan memastikan bahwa pengujian dilakukan dalam lingkungan yang mereplikasi produksi, sehingga meningkatkan kualitas perangkat lunak.
c. Peneliti: Memfasilitasi *reproducible* research adalah salah satu manfaat terbesar. Dengan kontainer, peneliti dapat mengemas lingkungan eksperimen mereka (termasuk kode, data, dan semua dependensi) ke dalam citra yang dapat dibagikan, memungkinkan peneliti lain mereplikasi hasil dengan presisi tinggi, meningkatkan transparansi dan validitas ilmiah [5].

Meskipun efisien dan fleksibel untuk skala yang dibahas, pendekatan berbasis skrip ini memiliki beberapa batasan intrinsik ketika dipertimbangkan untuk skala *enterprise* besar atau lingkungan produksi yang sangat kompleks dan kritis. Skrip ini tidak menyediakan fitur orkestrasi lanjutan seperti penyeimbangan beban otomatis antar kontainer, kemampuan auto-scaling (menyesuaikan jumlah kontainer berdasarkan beban), *self-healing* (secara otomatis mengganti kontainer yang gagal), manajemen jaringan yang canggih antar layanan multi-kontainer, atau sistem penyimpanan persisten terdistribusi yang mumpuni. Untuk kebutuhan yang melibatkan ribuan kontainer, manajemen klaster multi-node, aplikasi microservices yang kompleks, atau sistem yang sangat kritis, alat orkestrasi kontainer yang matang seperti *Kubernetes* atau *Docker Swarm* jauh lebih sesuai dan direkomendasikan [6]. *Kubernetes*, seperti yang diilustrasikan pada Gambar 4, menyediakan platform yang komprehensif dan powerful untuk manajemen siklus hidup aplikasi terkontainerisasi dalam skala besar, menawarkan abstraksi tingkat tinggi dan otomatisasi untuk kompleksitas infrastruktur.



Gambar 4. Arsitektur *Kubernetes*

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy
Formatted: Shadow

Gambar ini menunjukkan arsitektur dasar *Kubernetes*, platform orkestrasi kontainer terkemuka, yang mengelola *Node Worker* tempat *Pod* (kelompok kontainer) berjalan, diatur oleh *Control Plane*. Ini mencakup komponen seperti *Kubelet*, *Kube-proxy*, *etcd*, *API Server*, *Scheduler*, dan *Controller Manager* yang bekerja sama untuk mengelola klaster kontainer secara otomatis.

3.6 Diskusi Lanjutan dan Arah Penelitian Mendatang

Keberhasilan implementasi otomatisasi yang diuraikan dalam penelitian ini secara kuat menegaskan bahwa untuk skenario *on-demand* dengan skala terbatas yang mungkin berjalan pada satu atau beberapa *host* komputasi, pendekatan sederhana namun cerdas yang menggabungkan *Dockerfile* dan skrip shell sudah sangat memadai dan *cost-effective*. Solusi ini menawarkan alternatif yang jauh lebih ringan dan lebih cepat untuk diimplementasikan dibandingkan dengan investasi waktu dan sumber daya dalam mempelajari dan mengelola klaster *Kubernetes* yang kompleks, yang mungkin merupakan *overkill* untuk kebutuhan spesifik tersebut. Ini adalah solusi "pas" untuk banyak kasus penggunaan di mana infrastruktur minimalis diperlukan.

Namun, untuk pengembangan di masa depan dan untuk mengatasi beberapa batasan yang telah disebutkan, penelitian ini dapat diperluas ke beberapa arah yang menarik. Salah satunya adalah mengintegrasikan alat orkestrasi yang lebih ringan dan deklaratif, seperti *Docker Compose* [13], untuk mendefinisikan dan mengelola layanan multi-kontainer secara deklaratif melalui file *YAML* [14]. Pendekatan ini akan meningkatkan manajemen dependensi antar layanan yang berbeda dalam satu aplikasi, manajemen jaringan internal yang lebih rapi antar kontainer terkait, dan pengelolaan volume data persisten [9].

Arah lain adalah eksplorasi penggunaan configuration management *tools* seperti *Ansible*, *Chef*, atau *Puppet* [15]. Alat-alat ini dapat digunakan tidak hanya untuk mengotomatisasi *deployment* kontainer, tetapi juga untuk memusatkan dan mengamankan konfigurasi *host Docker* itu sendiri, memastikan bahwa semua *host Docker* dalam infrastruktur memiliki konfigurasi yang seragam dan aman sebelum kontainer di-*deploy* di atasnya [10]. Pendekatan ini akan meningkatkan robustess dan keamanan infrastruktur kontainer secara keseluruhan. Selain itu, penelitian di masa depan dapat mengeksplorasi integrasi dengan sistem monitoring dan logging terpusat untuk kontainer yang di-*deploy*, memberikan visibilitas yang lebih baik terhadap kinerja dan kesehatan aplikasi di dalam kontainer.

3.7 Perbandingan dengan Penelitian Sebelumnya

Hasil penelitian ini menunjukkan bahwa otomasi *deployment* berbasis *Docker* mampu menurunkan waktu *deployment* hingga 98,67% dibandingkan metode manual. Temuan ini mendukung penelitian Cherukuri (2024) [2] yang menegaskan bahwa *Docker* sebagai platform kontainerisasi mampu meningkatkan efisiensi dan konsistensi lingkungan pengembangan. Selain itu, hasil penelitian ini juga sejalan dengan Koneru (2025) [6] dan penelitian yang dipublikasikan di Frontiers in Robotics and AI (2024) [7], yang menekankan pentingnya peran *orchestration* tools seperti *Kubernetes* dan *Docker* Swarm dalam mendukung skalabilitas serta pengelolaan layanan pada skala besar.

Dibandingkan dengan penelitian Model-based tool (2024) [4] yang lebih menitikberatkan pada optimasi image untuk mengurangi ukuran dan waktu build, penelitian ini memberikan kontribusi tambahan berupa evaluasi kuantitatif terhadap efisiensi waktu *deployment*. Sementara itu, penelitian Mulpuri (2021) [8] menekankan otomasi *deployment* secara umum, penelitian ini lebih fokus pada implementasi skrip otomatis dan pengujian skenario *multi-container*. Dengan demikian, penelitian ini melengkapi studi-studi sebelumnya dengan memberikan bukti empiris mengenai peningkatan efisiensi waktu serta potensi penerapan pada skala *enterprise* melalui integrasi dengan *Kubernetes*.

4. KESIMPULAN

Penelitian ini telah berhasil merancang, mengimplementasikan, dan mengevaluasi secara komprehensif sebuah metodologi otomatisasi *deployment* dan manajemen multi-kontainer *Docker* yang sangat efisien untuk menciptakan lingkungan komputasi yang terisolasi dan scalable. Melalui integrasi yang cermat antara *Dockerfile* yang teroptimasi dan skrip shell yang dinamis, kami berhasil secara otomatis menyediakan dan mengkonfigurasi 20 kontainer Ubuntu secara efisien, masing-masing dilengkapi dengan akses pengguna *SSH* yang terpisah dan unik serta pemetaan *port* yang sistematis. Hasil eksperimen secara kuantitatif menunjukkan peningkatan efisiensi waktu *deployment* yang sangat signifikan, mencapai sekitar 98.67% pengurangan waktu dibandingkan dengan metode provisioning manual yang konvensional. Temuan ini secara tegas menegaskan bahwa pendekatan yang diusulkan sangat scalable untuk kasus penggunaan on-demand dan secara substansial mampu mengurangi overhead operasional. Selain itu, konsistensi konfigurasi antar kontainer dan integritas isolasi lingkungan telah diverifikasi secara menyeluruh, menjamin stabilitas, keamanan, dan reproduksibilitas.

Submitted: dd/mm/yyyy; Accepted: dd/mm/yyyy; Published: dd/mm/yyyy

Implikasi praktis dari penelitian ini memiliki jangkauan yang luas, meliputi peningkatan produktivitas yang substansial di lingkungan pendidikan, pengembangan perangkat lunak, dan pengujian sistem. Kemampuan untuk dengan cepat menyediakan sandbox yang terisolasi membuka peluang baru untuk pembelajaran kolaboratif dan pengujian yang efisien. Meskipun solusi ini sangat efektif untuk *deployment* pada skala *host* tunggal atau beberapa *host* secara independen, penting untuk dicatat bahwa terdapat tantangan untuk *deployment* skala *enterprise* besar yang membutuhkan orkestrasi canggih, seperti penyeimbangan beban otomatis, auto-scaling berdasarkan permintaan, kemampuan self-healing untuk kontainer yang gagal, dan manajemen penyimpanan persisten terdistribusi. Oleh karena itu, penelitian di masa mendatang dapat dan harus berfokus pada eksplorasi integrasi dengan alat orkestrasi yang lebih matang seperti *Docker Compose* untuk manajemen multi-layanan yang lebih terstruktur pada satu *host*, atau bahkan mengadopsi platform orkestrasi klaster yang lebih kuat seperti *Kubernetes* untuk mencapai *deployment* yang sangat scalable, fault-tolerant, dan siap produksi. Penelitian ini memberikan fondasi konseptual dan praktis yang kuat untuk pengembangan solusi otomatisasi infrastruktur kontainer yang lebih canggih dan adaptif di masa depan.

UCAPAN TERIMAKASIH

Terima kasih disampaikan kepada pihak-pihak yang telah mendukung terlaksananya penelitian ini, khususnya Program Studi Sistem Komputer, Universitas Pamulang Kota Serang untuk fasilitas laboratorium.

REFERENCES

cek turnitin261_PENELITIAN+TUNGGAL+HASAN+revission1.docx

ORIGINALITY REPORT

9%	9%	2%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

#	Source	Type	%
1	loddosinstitute.org	Internet Source	4%
2	ejournal.sisfokomtek.org	Internet Source	3%
3	pt.scribd.com	Internet Source	<1%
4	ojs.trigunadharma.ac.id	Internet Source	<1%
5	garuda.ristekbrin.go.id	Internet Source	<1%
6	journal.unilak.ac.id	Internet Source	<1%
7	accesson.kr	Internet Source	<1%
8	ejournal.undiksha.ac.id	Internet Source	<1%
9	ejurnal.seminar-id.com	Internet Source	<1%

10	ejournal.cvrobema.com Internet Source	<1 %
11	es.scribd.com Internet Source	<1 %
12	iswah.id Internet Source	<1 %
13	journal.uinsgd.ac.id Internet Source	<1 %
14	www.infokomputer.com Internet Source	<1 %
15	www.slideshare.net Internet Source	<1 %

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		