professional
training

Rilis Aplikasi Masih Terasa Lambat?

Cara Mengenali *Bottleneck* dan Mempercepat Delivery melalui CI/CD dengan Docker dan Jenkins

Dipersembahkan oleh
Btech Training
2026

Daftar Isi

Prolog

Rilis aplikasi yang lambat sering kali menjadi hambatan serius bagi perusahaan dalam menjaga daya saing. Masalah ini biasanya bersumber dari *bottleneck* pada proses pengembangan, mulai dari pengerjaan manual yang berulang hingga inkonsistensi lingkungan kerja. Akibatnya, *time-to-market* menjadi lama dan kepuasan pengguna pun menurun.

Untuk mengatasinya, diperlukan pendekatan yang terstandarisasi melalui penerapan CI/CD menggunakan Docker dan Jenkins. Dengan mengotomatiskan *pipeline* dan memastikan konsistensi sistem, perusahaan dapat mempercepat proses delivery, juga merespons kebutuhan pasar dengan jauh lebih gesit. Dokumen ini hadir untuk membantu Anda mewujudkannya.

Tentang Boer Technology

Sejak 2009, Boer Technology (Btech) telah berkomitmen kepada pelanggan dari berbagai industri di Indonesia untuk terus memberikan layanan dan solusi terbaik dengan menggunakan Perangkat Lunak Sumber Terbuka sebagai alat untuk menciptakan perusahaan yang ramah lingkungan. Didukung oleh sumber daya manusia yang andal, kami sekarang berfokus pada *Cloud*, *DevOps*, Keamanan, dan Pengiriman Profesional Teknologi untuk mewujudkan perusahaan yang efisien dan aman serta mempercepat jalannya bisnis.

Sejalan dengan komitmen tersebut, visi kami adalah menjadi penyedia solusi terbaik di bidang *Managed Services* dan Sistem Integrasi *cloud* yang inovatif, mendunia, dan mendorong pertumbuhan talenta IT Indonesia.

Layanan Kami

Managed Services

Layanan kami menangani operasional harian infrastruktur Anda, memungkinkan tim Anda untuk fokus pada percepatan pertumbuhan bisnis. Dengan pemantauan 24/7 oleh para tenaga ahli berpengalaman kami, kami memastikan keandalan berkelanjutan dan memberikan ketenangan pikiran yang sesungguhnya.

Maintenance Support

Dukungan pemeliharaan diberikan melalui sistem berbasis tiket, di mana kami siap siaga sesuai dengan tingkat layanan yang disepakati untuk menangani insiden dan permintaan operasional. Didukung oleh para insinyur berpengalaman, kami memastikan masalah ditangani secara efisien untuk menjaga keandalan lingkungan *cloud* Anda.

Implementation

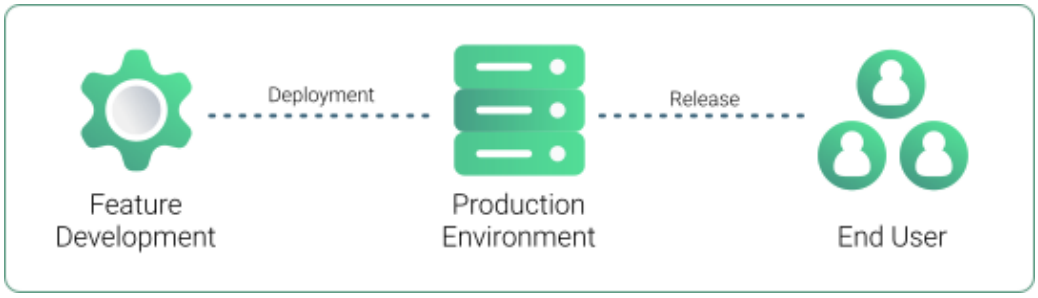
Layanan ini berfokus pada pelaksanaan solusi berdasarkan desain dan persyaratan yang telah disetujui, memastikan proses penerapan yang lancar dan terkontrol. Dengan masa garansi pasca-implementasi, kami memberikan jaminan dan dukungan tambahan untuk melindungi sistem Anda setelah beroperasi.

Consultation

Kami membantu perusahaan mengatasi tantangan infrastruktur TI yang kompleks dengan menerapkan praktik terbaik yang telah terbukti dan selaras dengan tujuan bisnis. Melalui dokumentasi yang jelas dan peta jalan strategis, kami memberikan panduan terstruktur yang disesuaikan dengan tujuan Anda.

Bab 1 - Kecepatan Rilis Fitur

Mengapa Time-to-Market Penting untuk Daya Saing Bisnis Anda?



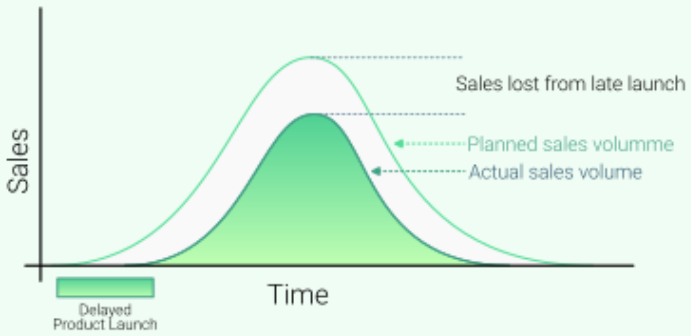
Gambar 1.1. Ilustrasi rilis fitur dan percepatan *time-to-market*

Dalam bisnis digital yang dinamis sekarang, kemampuan untuk merilis fitur dengan cepat tidak lagi sekadar keuntungan tambahan, melainkan sebuah prasyarat esensial bagi kelangsungan dan kesuksesan perusahaan di pasar yang kompetitif. Di mana tim yang mampu menghadirkan inovasi dan perubahan secara gesit cenderung lebih efektif dalam memahami aspirasi pelanggan, menguji hipotesis baru, serta sigap memanfaatkan peluang yang muncul.

Meskipun demikian, seringkali kita melihat tim IT yang berdedikasi tinggi justru menghadapi tantangan yaitu seperti contoh proses rilis sebuah fitur, bahkan yang terkecil sekalipun, ke lingkungan hasil dapat memakan waktu berhari-hari, bahkan hingga berminggu-minggu.

Menurut penelitian terbaru, praktik CI/CD berkorelasi dengan peningkatan kecepatan delivery sekaligus kualitas rilis karena proses integrasi, pengujian, dan *deployment* menjadi lebih terstruktur (Shahin et al., 2017).

Apa itu *Time-to-Market*?



Gambar 1.1. Definisi *time-to-market*

Time-to-market menggambarkan durasi yang diperlukan, mulai dari munculnya ide atau keputusan bisnis hingga fitur produk tersedia secara langsung dan dapat diakses oleh pelanggan. Kepenuhan dalam menghadirkan produk ke pasar menjadi faktor krusial bagi peningkatan peluang pertumbuhan bisnis. Penundaan dalam peluncuran berpotensi mengurangi *Planned Sales Volume* (Target penjualan) yang telah Anda targetkan, menjadikannya *Actual Sales Volume* (Target aktual) yang lebih rendah dari yang diharapkan. Konsekuensinya, akan timbul *Sales Lost* (lenyapnya potensi laba yang sifatnya permanen), sebab momentum pasar yang telah hilang tak akan pernah bisa direbut kembali.

1.1 Ketika Bisnis Bergerak Cepat, tetapi Tim IT Tertahan

Gambar 1.2. Ilustrasi hambatan saat tim engineering tertahan oleh proses rilis

Coba bayangkan skenario di mana tim produk telah menyelesaikan risetnya, dan para pemangku kepentingan telah memberikan persetujuan, dan rancang desain pun sudah final. Namun, ketika giliran tim teknik untuk mengerjakan fitur tersebut justru terhenti. Hal ini bukan karena kurangnya kapabilitas tim, melainkan akibatnya kendala dalam alur kerja rilis fitur nya. Situasi semacam ini bukanlah hal baru. Seringkali, akar permasalahan utamanya tidak terletak pada kompetensi individu, melainkan pada kurangnya efisiensi dalam proses yang berjalan.

Implikasi dari keterlambatan rilis ini, yang kerap terlewatkan, meliputi beberapa aspek berikut:

- Fitur yang dirancang untuk menjadi keunggulan kompetitif justru baru tersedia setelah pesaing Anda lebih dulu mengimplementasikannya di pasar.
- Perbaikan atas *bug* krusial menjadi tertunda, entah karena padatnya jadwal penyebaran atau karena rumitnya prosedur yang harus dilalui.
- Semangat tim *engineering* berpotensi menurun drastis karena hasil kerja keras mereka membutuhkan waktu yang sangat panjang untuk sampai ke tangan pengguna akhir.
- Secara tidak disadari, biaya operasional dapat membengkak akibat keterlibatan langkah-langkah manual yang repetitif dalam jumlah berlebihan.

Studi multi-perusahaan menunjukkan bahwa tim yang mengadopsi *DevOps* secara serius mengalami peningkatan kecepatan rilis dan koordinasi lintas fungsi yang lebih baik (Lwakatare et al., 2019).

1.2 Mengapa Proses Manual Tidak Bisa Terus Dipertahankan?

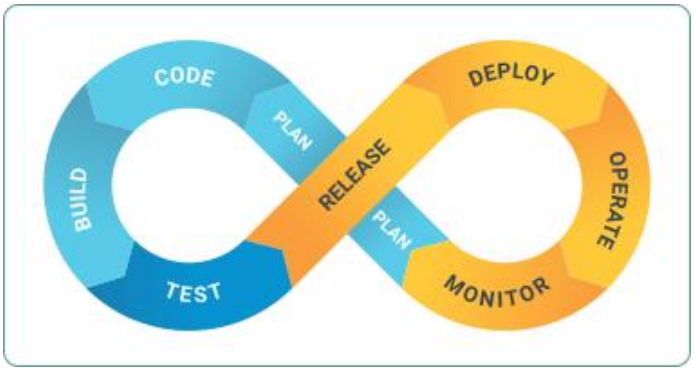
Ketika skala tim masih kecil, operasional manual mungkin masih terasa cukup terkendali. Namun, seiring dengan pertumbuhan jumlah insinyur, peningkatan kompleksitas aplikasi, dan tuntutan bisnis untuk frekuensi rilis yang lebih tinggi, metode kerja manual ini perlahan-lahan mulai menjelma menjadi sebuah beban.

Indikator-indikator berikut mungkin menunjukkan bahwa proses manual Anda mulai menghambat laju tim:

- *Engineer* senior Anda lebih banyak menghabiskan waktu mengawasi proses *deployment*, ketimbang fokus pada tugas-tugas yang memiliki nilai strategi tinggi.
- Setiap upaya rilis terasa begitu menegangkan, sebab memerlukan verifikasi manual pada serangkaian langkah yang kompleks dan berpotensi menimbulkan kesalahan.
- Kurangnya standardisasi dalam proses rilis, jadi pada setiap individu cenderung menerapkan pendekatan yang berbeda satu sama lain.
- Anggota tim yang baru bergabung membutuhkan waktu signifikan hanya untuk menguasai alur *deployment* yang ada.

> Kajian literatur *continuous delivery* menunjukkan hambatan ini konsisten muncul di banyak organisasi, terutama pada area *test automation*, dependensi proses manual, dan koordinasi lintas tim (Laukkanen et al., 2017).

1.3 CI/CD Sebagai Jawaban



Gambar 1.3. Ilustrasi CI/CD sebagai jawaban untuk mempercepat *delivery*

Penerapan *Continuous Integration* (CI) dan *Continuous Delivery* (CD) dirancang untuk membantu tim mengotomatisasi seluruh siklus alur kerja, mulai dari penulisan kode hingga kesiapan untuk dirilis. Melalui optimalisasi proses yang lebih terstruktur dan konsisten, tim dapat mengalihkan fokus dan energinya secara penuh pada inovasi pengembangan produk, alih-alih terbebani oleh tugas-tugas rilis yang repetitif.

Pada bagian selanjutnya, kita akan mengeksplorasi strategi untuk mengidentifikasi hambatan (*bottleneck*), mendalami fungsi krusial dari Docker dan Jenkins, serta memandu Anda dalam merancang *pipeline* pertama yang praktis dan sesuai untuk kebutuhan tim Anda.

Catatan untuk *Decision Maker*:

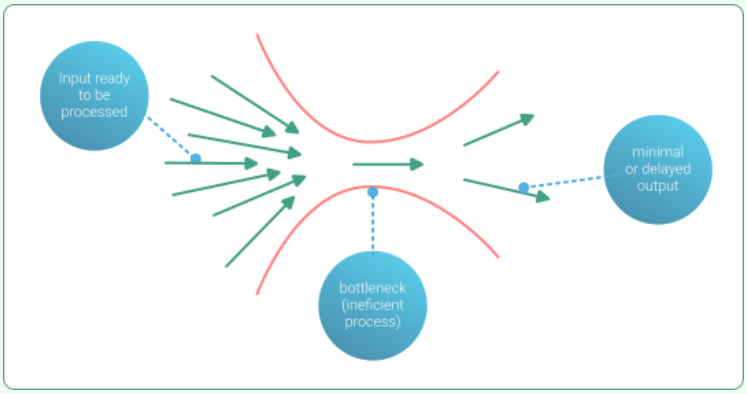
Investasi CI/CD umumnya mulai terasa dalam beberapa bulan awal melalui penurunan *error* manual, peningkatan kecepatan rilis, dan visibilitas proses yang lebih baik (Amaro et al., 2023).

Bab 2 - Mengidentifikasi *Bottleneck* di Proses Rilis

Sebelum memperbaiki proses, Anda perlu melihat letak masalahnya dengan jelas.

Seringkali, banyak tim beranggapan bahwa alur rilis mereka sudah memadai, hingga mereka menyadari bahwa sebuah perubahan minor / kecil memerlukan waktu hingga tiga hari untuk dapat diterbitkan di lingkungan produksi. Lazimnya, perlambatan ini bukan disebabkan oleh kualitas kode itu sendiri, melainkan oleh sejumlah hambatan kecil yang terakumulasi di sepanjang siklus proses.

Apa yang dimaksud *bottleneck*?



Gambar 2.1. Ilustrasi *Bottleneck*

Bottleneck adalah titik dalam alur kerja yang paling memperlambat keseluruhan proses. Dalam proses rilis *software*, satu hambatan kecil saja bisa membuat tahapan lain yang sudah cepat ikut tertahan.

2.1 Tanda-tanda Adanya *Bottleneck* di Tim Anda

Hambatan atau *bottleneck* ini sering kali tidak terdeteksi karena sudah dianggap sebagai bagian integral dari rutinitas operasional tim. Namun, ada beberapa indikator yang cukup jelas yang bisa menjadi sinyal peringatan:

- Pengujian masih sepenuhnya dilakukan secara manual setiap kali terjadi perubahan pada kode.
- Hanya satu atau dua individu yang memiliki akses atau pengetahuan yang diperlukan untuk melakukan *deployment*.

- Tim perlu menunggu persetujuan atau konfirmasi dari berbagai fungsi terkait sebelum dapat melanjutkan ke tahap berikutnya.
- Terdapat perbedaan konfigurasi antara lingkungan *development*, *staging*, dan *production*, yang berujung pada munculnya *bug* baru ketika aplikasi dipindahkan ke lingkungan yang berbeda.
- Tidak tersedia riwayat rilis yang terdokumentasi dengan baik mengenai siapa yang melakukan rilis, kapan rilis dilakukan, dan versi apa yang diimplementasikan.

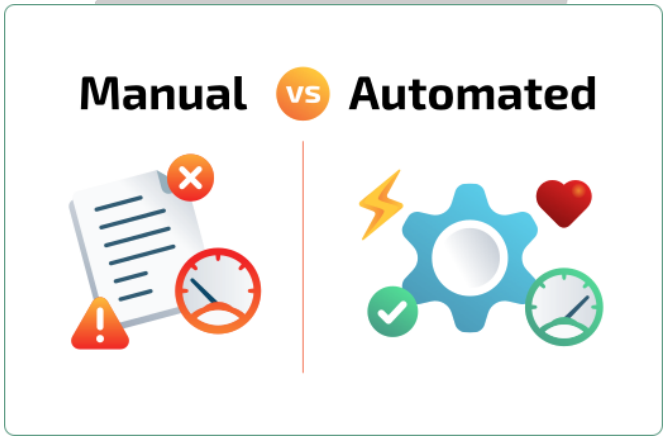
2.2 Tiga Jenis *Bottleneck* Paling Umum

a) *Bottleneck* Manusia (*Human Dependency*)

Jenis hambatan ini terjadi ketika sebuah proses terlalu mengandalkan individu atau peran tertentu. Sebagai contoh, hanya seorang *engineer* senior yang diizinkan untuk melakukan *merge* ke *branch* utama, atau hanya satu individu yang memegang kendali akses ke *server* produksi. Ketika individu tersebut tidak tersedia, alur kerja dapat terhenti secara signifikan.

Pendekatan solutifnya bukan menghilangkan peran krusial mereka, melainkan memastikan bahwa pengetahuan, akses, dan langkah-langkah kerja dapat didistribusikan dan dieksekusi secara lebih merata melalui penerapan otomatisasi.

b) *Bottleneck* Proses (*Manual Workflow*)



Gambar 2.1. Manual vs *Automation*

Semakin banyak tahapan yang dilakukan secara manual, semakin tinggi potensi terjadinya perlambatan dalam proses. Mulai dari proses *build* aplikasi, pemindahan *file* ke *server*, hingga pemberitahuan kepada tim QA melalui komunikasi digital, semua ini berpotensi menjadi titik *bottleneck* dan sumber kekeliruan.

c) *Bottleneck* Lingkungan (*Environment Inconsistency*)

Perbedaan konfigurasi antar lingkungan adalah tantangan yang umum dijumpai. Aplikasi mungkin berfungsi dengan lancar di perangkat *developer*, namun gagal ketika dipindahkan ke lingkungan *staging* atau produksi, sering kali hanya karena perbedaan versi *library* atau variabel lingkungan. Pada titik inilah, Docker menawarkan solusi yang signifikan.

> Temuan empiris menunjukkan bahwa lamanya *build CI* dan antrian *pipeline* adalah penyebab nyata lambatnya *feedback loop*, yang akhirnya memperpanjang *lead time* (Ghaleb et al., 2019).

2.3 Cara Mengukur *Bottleneck* Secara Objektif

Demi mencapai penilaian yang lebih objektif, identifikasi hambatan atau *bottleneck* sebaiknya didasarkan pada data konkret, bukan hanya asumsi atau persepsi semata. Disarankan untuk memulai dengan memantau metrik berikut secara berkala.

> Penelitian tentang kapabilitas dan metrik *DevOps* menegaskan bahwa organisasi dengan disiplin pengukuran yang baik cenderung lebih cepat melakukan perbaikan proses secara berkelanjutan (Amaro et al., 2023).

Table 2.1. Metrik untuk mengukur *bottleneck* secara objektif

Metrik	Definisi	Target Sehat
Lead Time	Waktu dari perubahan pertama sampai fitur *live* di *production*	Idealnya kurang dari 1 hari untuk perubahan kecil
Deployment Frequency	Seberapa sering tim merilis ke *production* dalam seminggu atau sebulan	Tim sehat minimal 1x per minggu
Change Failure Rate	Persentase rilis yang menimbulkan gangguan atau *rollback* di *production*	Target di bawah 15 persen

Apabila *lead time* untuk perubahan minor masih melebihi tiga hari, atau frekuensi *deployment* belum mencapai setidaknya satu kali dalam seminggu, maka ada indikasi kuat bahwa terdapat *bottleneck* yang memerlukan perbaikan segera.

2.4 Audit Singkat Untuk Mengecek Kondisi Tim Anda

Jawablah pertanyaan-pertanyaan berikut dengan objektif. Semakin banyak respons 'Ya' yang Anda berikan, semakin jelas pula bahwa proses rilis tim Anda masih memiliki hambatan yang perlu segera diatasi.

Table 2.2. *Checklist* audit singkat kondisi proses rilis tim

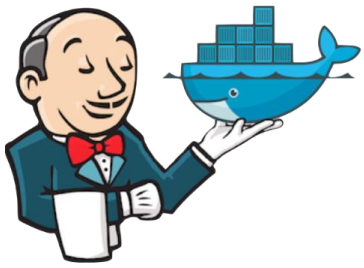
Pertanyaan	Ya	Tidak
Apakah proses *build* masih dijalankan manual oleh *engineer*?	☐	☐
Apakah *testing* masih membutuhkan intervensi manual setiap ada perubahan kode?	☐	☐
Apakah *deployment* hanya bisa dilakukan oleh orang tertentu?	☐	☐
Apakah konfigurasi *production* masih berbeda dengan *development* atau *staging*?	☐	☐
Apakah belum ada *pipeline* otomatis yang menjalankan *test* setelah *commit*?	☐	☐

Catatan untuk Tim *Lead*

Audit singkat ini akan paling efektif jika digunakan sebagai landasan diskusi tim. Arahkan pembahasannya pada upaya perbaikan proses, dan hindari mencari siapa yang bertanggung jawab atas masalah yang ada.

Bab 3 - Docker & Jenkins

Standarisasi dan otomatisasi sebagai solusi yang relevan untuk bisnis.



Gambar 3.1. Ilustrasi Docker dan Jenkins sebagai fondasi otomatisasi

Setelah mengidentifikasi titik-titik hambatan, langkah selanjutnya adalah memilih perangkat yang benar-benar mendukung dan efektif dalam melakukan otomatisasi rilis fitur. Dalam berbagai diskusi mengenai CI/CD, Docker dan Jenkins kerap menjadi topik utama, mengingat keduanya bukan hanya sekadar alat teknis, melainkan juga pendorong kapabilitas operasional yang esensial bagi bisnis.

3.1 Docker: Solusi untuk Lingkungan *Inconsistency*

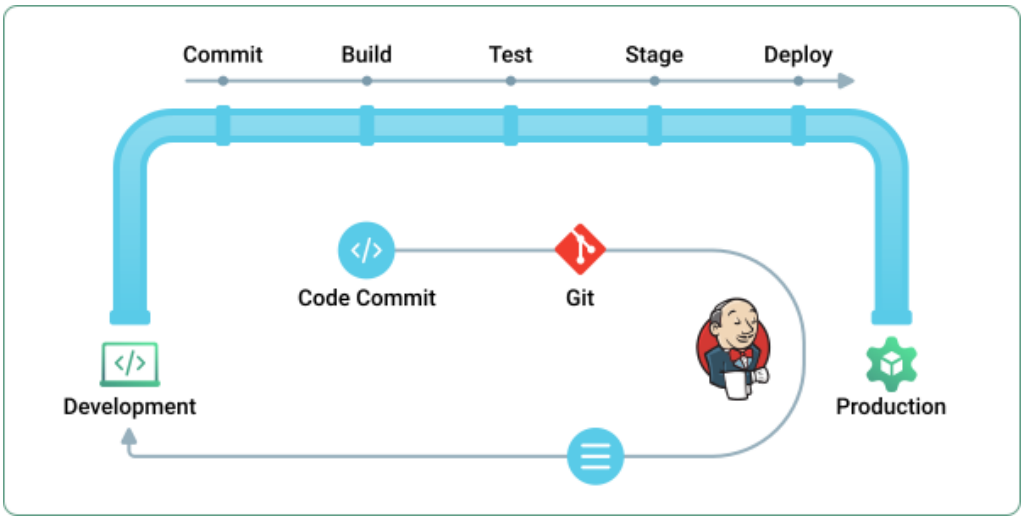
Docker merupakan sebuah platform *containerization* yang berfungsi untuk mengemas aplikasi beserta seluruh dependensinya ke dalam sebuah tempat atau unit-unit terisolasi yang disebut sebagai *container*. Dengan demikian, aplikasi dapat dieksekusi dengan perilaku yang konsisten, baik di lingkungan *developer*, *server staging*, maupun lingkungan *production*.

Manfaat Docker bagi bisnis umumnya dapat diamati pada aspek-aspek berikut:

- Isu *"works on my local machine"* berkurang secara signifikan berkat konsistensi lingkungan kerja yang terwujud sejak awal.
- *Developer* baru dapat melakukan proses *onboarding* lebih efisien, sebab tidak memerlukan waktu yang lama untuk melakukan *setup* lingkungan kerja.
- Proses *rollback* menjadi lebih sederhana, karena setiap versi aplikasi disimpan dalam bentuk *image* yang terpisah dan terisolasi.
- Skalabilitas menjadi lebih adaptif, mengingat *container* dapat dijalankan dan direplikasi sesuai dengan tuntutan kebutuhan.

Evaluasi performa Docker lintas sistem operasi juga memperlihatkan ada variasi kinerja yang perlu dipertimbangkan saat menentukan *environment build deploy* (Sobieraj & Kotyński, 2024).

3.2 Jenkins: Otomatisasi *Pipeline* dari Ujung ke Ujung



Gambar 3.2. Ilustrasi Jenkins sebagai *automation server* untuk *pipeline* CI/CD

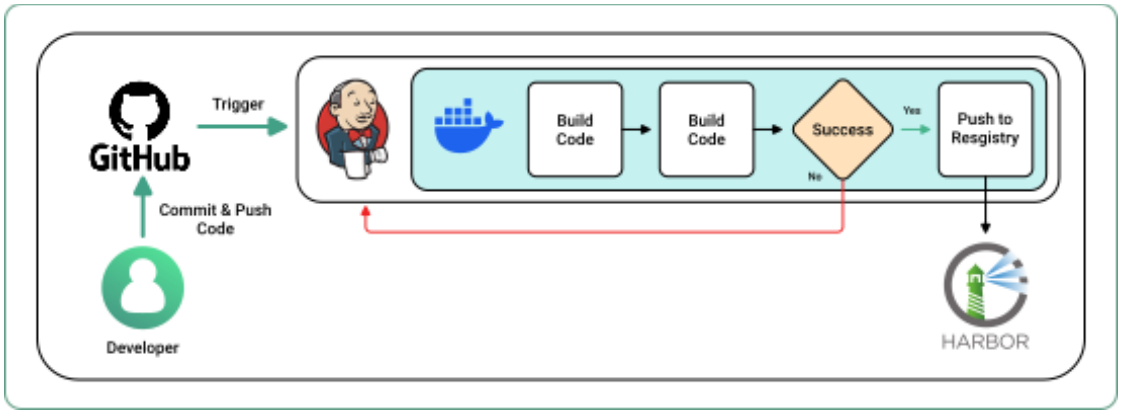
Jenkins adalah *automation server open-source* yang secara luas digunakan untuk mengelola dan mengeksekusi *pipeline* CI/CD. CI sendiri kepanjangan dari *Continuous Integration* yang di mana saat ingin merilis fitur, akan otomatis *merge branch* utama jika ada perubahan, serta memungkinkan perbaikan *bug* menjadi lebih cepat, lalu untuk CD adalah *Continuous Delivery*, yang di mana berfungsi untuk *build* serta melakukan *testing code* secara otomatis, kemudian CD singkatan dari *Continuous Deployment* yang di mana akan otomatis ter-*deploy* ke dalam *server staging* atau bahkan sampai ke *production* tanpa campur tangan manusia.

Melalui Jenkins, proses yang mencakup *commit*, pengujian, *build*, hingga *deployment* dapat diotomatisasi dengan alur kerja yang lebih terstruktur dan terkontrol. Manfaat yang ditawarkan Jenkins bagi bisnis dapat dirasakan dalam beberapa aspek berikut:

- Setiap tindakan *commit* dapat secara otomatis memicu proses *build* dan pengujian, tanpa memerlukan intervensi manual.
- Tim akan menerima notifikasi lebih cepat saat terdapat kegagalan pada tahap *pipeline*, sehingga memungkinkan respon yang lebih sigap.
- Jejak audit menjadi lebih transparan dan informatif: mencakup informasi mengenai siapa yang menginisiasi *pipeline*, kapan eksekusi dilakukan, dan bagaimana hasil akhirnya.
- Jenkins mudah diintegrasikan dengan berbagai perangkat lain seperti Git, Slack, Docker *Registry*, serta beragam layanan *cloud*.

Contoh pemakaian Jenkins pada konteks industri menunjukkan *platform* ini efektif sebagai orkestrator proses berulang dan terstandar (Moutsatsos et al., 2017).

3.3 Bagaimana Docker dan Jenkins Bekerja Bersama



Gambar 3.3. Ilustrasi kolaborasi Docker dan Jenkins dalam *pipeline* CI/CD

Docker dan Jenkins beroperasi secara sinergis untuk memperkuat satu sama lain. Jenkins berperan dalam mengelola alur kerja dan otomatisasi, sementara Docker memastikan konsistensi lingkungan eksekusi di setiap fase *pipeline*.

Secara ringkas, alurnya seperti ini:

1. *Developer* melakukan *commit* ke Git *repository*.
2. Jenkins mendeteksi perubahan dan menjalankan *pipeline* secara otomatis.
3. Jenkins mengeksekusi testing di dalam *container* Docker yang bersih dan terisolasi.
4. Jika seluruh test lolos, Jenkins membangun Docker *image* yang baru.
5. *Image* dikirim ke *registry* dan siap digunakan untuk *deployment*.

Setelah implementasi dasar berjalan, ada beberapa aspek yang layak dievaluasi agar manfaat CI/CD benar-benar terasa dan mudah diaudit secara singkat.

> Kajian arsitektur *continuous delivery* menekankan bahwa desain sistem dan *pipeline* harus mendukung *unit deployment* yang kecil, terisolasi, dan mudah diuji agar delivery benar-benar cepat (Shahin et al., 2018).

Table 3.1. Aspek evaluasi setelah implementasi CI/CD

Aspek Evaluasi	Yang Dicek	Indikator Cepat
Keandalan *pipeline*	Seberapa sering *pipeline* sukses dari *commit* sampai *build* atau *deploy*	Sebagian besar *run* selesai tanpa perbaikan manual berulang
Kecepatan rilis	Waktu dari *commit* sampai *deployment* siap pakai	*Lead time* turun dan antrean rilis berkurang
Konsistensi *environment*	Kesesuaian hasil di *development*, *staging*, dan *production*	Kasus *works on my machine* makin jarang
Kualitas dan *traceability*	Ketersediaan *log*, hasil *test*, dan riwayat perubahan	Mudah menelusuri siapa, kapan, dan versi apa yang dirilis
Efisiensi tim	Waktu *engineer* yang tersita untuk langkah operasional	Lebih sedikit *copy-paste*, *approval* manual, dan pengecekan ulang

Dari sisi proses, perbandingan berikut memperlihatkan langkah-langkah manual yang biasanya paling banyak dipangkas saat CI/CD sudah berjalan.

Table 3.2. Perbandingan proses manual dan CI/CD

Parameter Perbandingan	Metode Manual	Implementasi CI/CD (Otomatis)	Dampak/ *Impact* (%)
Kecepatan & Frekuensi *Deployment*	*Deployment* dilakukan berkala dengan yang memakan waktu lama.	*Deployment* dilakukan secara kontinu.	Peningkatan frekuensi hingga 200x lipat.
Lead Time for Changes	Membutuhkan waktu berminggu-minggu karena verifikasi manual.	hitungan jam atau menit melalui *pipeline* otomatis.	Pengurangan waktu rilis rata-rata sebesar 60%.
Change Failure Rate (Tingkat Kegagalan)	Memiliki risiko kegagalan tinggi (20-30%) akibat kesalahan manusia (*human error*).	Meminimalisir kegagalan berkat pengujian otomatis yang ketat.	Penurunan tingkat kegagalan rilis sebesar 70-80%.
Mean Time to Recovery (MTTR)	Pemulihan sistem saat *error* membutuhkan waktu berjam-jam.	Pemulihan dilakukan secara instan dengan *automated* dalam hitungan menit.	Waktu pemulihan lebih cepat hingga 45-92%.

Secara praktik, implementasi *DevOps* yang matang terbukti meningkatkan hasil operasional lintas tim dan membantu menyeimbangkan *speed vs stability* (Lwakatare et al., 2019).

Bab 4 - Langkah Pertama Membangun *Pipeline* CI/CD

Dari nol sampai pipeline pertama Anda berjalan.

Memulai CI/CD tidak harus selalu rumit seperti yang banyak dibayangkan orang-orang. *Pipeline* yang sederhana tetapi stabil jauh lebih berguna daripada *pipeline* kompleks yang tidak pernah benar-benar dipakai. Fokus utama tahap awal adalah membangun fondasi yang bisa diandalkan.

4.1 Prasyarat Sebelum Memulai

Sebelum membuat *pipeline*, pastikan komponen berikut sudah tersedia di lingkungan kerja Anda:

- *Git Repository* seperti *GitHub*, *GitLab*, atau *Bitbucket*. *Pipeline* akan di-*trigger* dari sini.
- Jenkins *Server* yang dapat dipasang di *VM*, *cloud*, atau dijalankan sebagai *container* Docker.
- Docker yang sudah terpasang di *server* Jenkins dan di *server* tujuan *deployment*.
- *Dockerfile* yang menjelaskan bagaimana *image* aplikasi Anda dibangun.

4.2 Struktur *Pipeline* CI/CD Dasar

Pipeline CI/CD yang baik punya alur yang jelas dan berurutan. Setiap tahap memegang tanggung jawab tertentu, dan jika satu tahap gagal, *pipeline* sebaiknya berhenti agar masalah tidak ikut terbawa ke tahap berikutnya.

Empat tahap utama dalam *pipeline* sederhana adalah:

1. *Checkout* mengambil kode terbaru dari repositori *Git*.
2. *Build* membuat Docker *image* dari kode yang sudah lolos pengujian.
3. *Push Images* ke *Container registry* setelah hasil build berhasil tanpa *error*.
4. *Deploy* menjalankan *container* baru dari *image* yang baru saja dibuat.

4.3 Contoh *Jenkinsfile* (*YAML-style Declarative Pipeline*)

Berikut adalah *Link* contoh *Jenkinsfile* dengan format *Declarative Pipeline* yang sederhana dan mudah dipahami. *File* ini disimpan di *root* repositori Anda dengan nama *Jenkinsfile*.

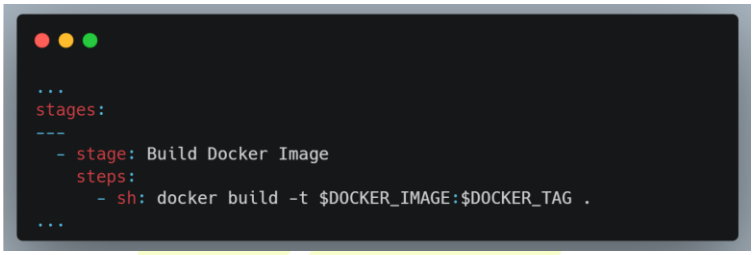
4.4 Memahami Setiap Blok *Pipeline*

Environment

Gambar 4.1. Ilustrasi blok *environment* pada Jenkins *pipeline*

Blok *environment* berisi variabel yang dipakai di seluruh *pipeline*. Dengan cara ini, perubahan konfigurasi cukup dilakukan di satu tempat.

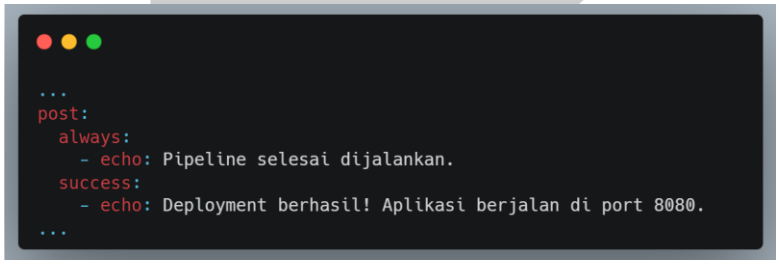
Stages



Gambar 4.2. Ilustrasi susunan *stages* dalam *pipeline* CI/CD

Stage adalah tahapan kerja di dalam *pipeline.* Jenkins akan mengeksekusinya secara berurutan. Jika satu stage gagal, tahap setelahnya tidak dijalankan sehingga kode bermasalah tidak ikut melaju ke *production*.

Post Actions

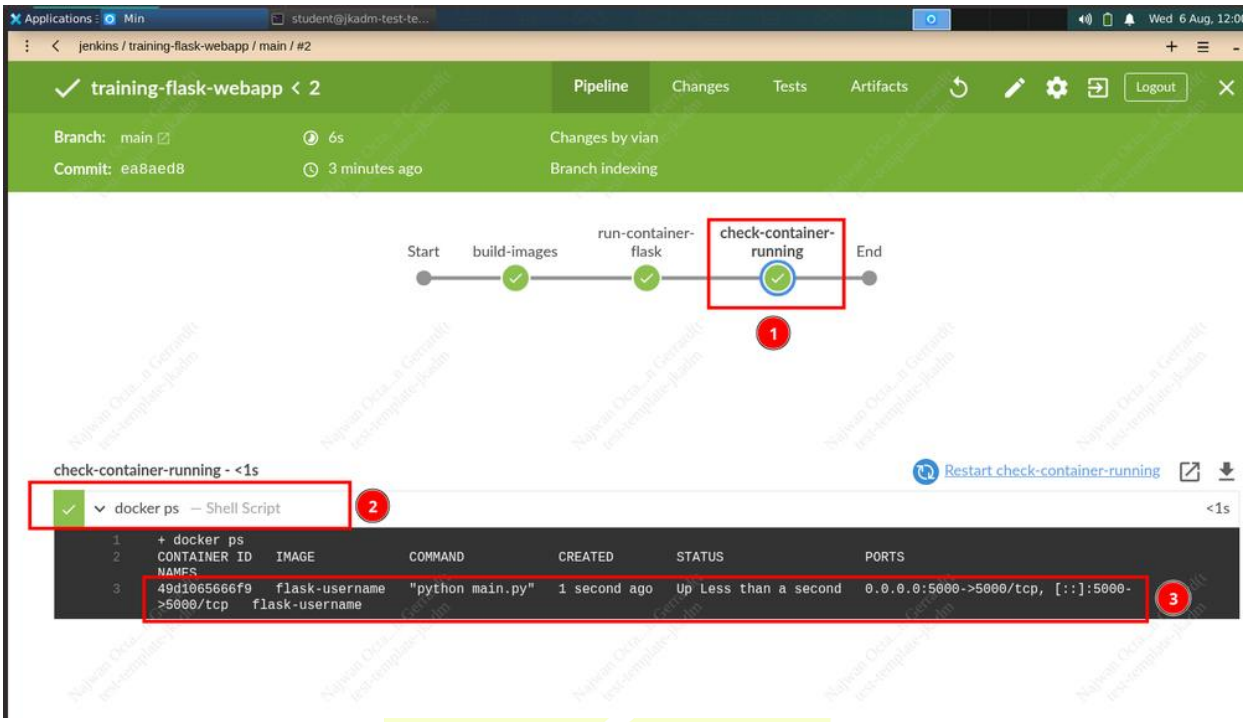


Gambar 4.3. Ilustrasi *post actions* untuk notifikasi dan *cleanup*

Blok post dijalankan setelah seluruh *stage* selesai, apa pun hasilnya. Bagian ini cocok dipakai untuk mengirim notifikasi, membersihkan *resource* sementara, atau menjalankan langkah penutup lain.

> Penelitian terbaru tentang pengujian *NFR* di *CI* menunjukkan bahwa kualitas *pipeline* meningkat signifikan ketika *quality gate* tidak hanya fokus pada tes fungsional, tapi juga aspek non-fungsional seperti *reliability/performance* sesuai konteks bisnis (Yu et al., 2023).

4.5 Sample Hasil CI/CD *Pipeline*



Gambar 4.4 *Sample Logs Pipeline* Jenkins

Di atas adalah salah satu contoh *pipeline* dari *Jenkins*, yang di mana pada *pipeline* tersebut melakukan *build* aplikasi menjadi *images Docker*, kemudian menjalankan aplikasi tersebut di dalam *Docker container*, serta di bagian akhir *pipeline* ada step untuk pengecekan hasil *container* yang sudah dibuat.

Bab 5 - Mengubah *Downtime* Menjadi *Productive Time*

Perubahan yang biasanya paling terasa setelah CI/CD *berjalan.*

Ketika *pipeline* CI/CD sudah aktif, dampaknya tidak berhenti di kecepatan *deployment* saja. Tim biasanya juga merasakan perubahan dalam cara bekerja, cara menjaga kualitas, dan cara memanfaatkan waktu.

5.1 Apa yang Berubah Setelah CI/CD Berjalan

Waktu *deployment* turun drastis
Pekerjaan yang sebelumnya memakan 2-3 jam secara manual sering kali bisa selesai dalam 5-15 menit. *Engineer* pun tidak perlu lagi berjaga hanya untuk memastikan proses rilis berjalan sampai akhir.

Jumlah *human error* berkurang
Kesalahan seperti lupa menjalankan perintah, salah urutan langkah, atau konfigurasi yang terlewat jauh lebih jarang terjadi karena *pipeline* yang sama dijalankan secara konsisten setiap kali.

Kepercayaan diri tim meningkat
Tim yang tadinya ragu saat merilis biasanya menjadi lebih tenang, karena pengujian otomatis sudah berjalan lebih dulu. Dampaknya bukan hanya teknis, tetapi juga terasa pada budaya kerja tim.

5.2 *Engineer* Bisa Fokus pada Hal yang Benar-Benar Penting

Salah satu manfaat besar CI/CD adalah mengembalikan waktu *engineer* ke pekerjaan yang memang membutuhkan pemikiran manusia. Waktu yang sebelumnya habis untuk rutinitas manual bisa dialihkan ke hal-hal berikut:

- Mencari pendekatan teknis yang lebih baik dan relevan untuk produk.
- Melakukan *code review* yang lebih tajam dan bermanfaat.
- Menyusun dokumentasi yang benar-benar membantu tim.
- Menguji teknologi baru yang masuk akal untuk kebutuhan bisnis.

5.3 Dari Perspektif Bisnis

Bagi *decision maker*, dampak CI/CD biasanya lebih mudah dipahami jika diterjemahkan ke hasil operasional. Walau tiap tim bisa berbeda, pola berikut cukup sering muncul setelah implementasi berjalan:

- Frekuensi rilis meningkat; tim yang tadinya merilis bulanan sering bisa bergerak ke mingguan, bahkan harian.
- Waktu pemulihan lebih cepat; saat ada masalah di *production*, rollback dapat dilakukan dalam hitungan menit.
- Kepuasan tim naik; *engineer* yang tidak terbebani proses manual biasanya lebih fokus dan lebih produktif.
- Kepercayaan *stakeholder* ikut tumbuh karena rilis menjadi lebih konsisten dan stabil.

Studi desain-sains tentang kapabilitas *DevOps* menunjukkan penguatan proses, metrik, dan pembelajaran tim berkelanjutan merupakan faktor kunci yang memengaruhi kinerja organisasi (Amaro et al., 2023). Kajian literatur terbaru juga memetakan kapabilitas *DevOps* di seluruh siklus hidup *software* sehingga *roadmap* peningkatan bisa dibuat lebih sistematis (Amaro et al., 2025).

Daftar Referensi Jurnal

1. Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, 5, 3909-3943. https://doi.org/10.1109/ACCESS.2017.2685629

2. Laukkanen, E., Itkonen, J., & Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery-A systematic literature review. Information and Software Technology, 82, 55-79. https://doi.org/10.1016/j.infsof.2016.10.001

3. Shahin, M., Zahedi, M., Babar, M. A., & Zhu, L. (2018). An empirical study of architecting for continuous delivery and deployment. Empirical Software Engineering, 24(3), 1061-1108. https://doi.org/10.1007/s10664-018-9651-4

4. Ghaleb, T. A., da Costa, D. A., & Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. Empirical Software Engineering, 24(4), 2102-2139. https://doi.org/10.1007/s10664-019-09695-9

5. Lwakatare, L. E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M., & Lassenius, C. (2019). DevOps in practice: A multiple case study of five companies. Information and Software Technology, 114, 217-230. https://doi.org/10.1016/j.infsof.2019.06.010

6. Amaro, R., Pereira, R., & da Silva, M. M. (2023). Capabilities and metrics in DevOps: A design science study. Information & Management, 60(5), 103809. https://doi.org/10.1016/j.im.2023.103809

7. Amaro, R., Pereira, R., & da Silva, M. M. (2025). Mapping DevOps capabilities to the software life cycle: A systematic literature review. Information and Software Technology, 177, 107583. https://doi.org/10.1016/j.infsof.2024.107583

8. Fernández González, D., Rodríguez Lera, F. J., Esteban, G., & Fernández Llamas, C. (2021). SecDocker: Hardening the Continuous Integration Workflow. SN Computer Science, 3(1). https://doi.org/10.1007/s42979-021-00939-4

9. Azuma, H., Matsumoto, S., Kamei, Y., & Kusumoto, S. (2022). An empirical study on self-admitted technical debt in Dockerfiles. Empirical Software Engineering, 27(2). https://doi.org/10.1007/s10664-021-10081-7

10. Yu, L., Alégroth, E., Chatzipetrou, P., & Gorschek, T. (2023). Automated NFR testing in continuous integration environments: a multi-case study of Nordic companies. Empirical Software Engineering, 28(6). https://doi.org/10.1007/s10664-023-10356-1

11. Sobieraj, M., & Kotyński, D. (2024). Docker Performance Evaluation across Operating Systems. Applied Sciences, 14(15), 6672. https://doi.org/10.3390/app14156672

12. Moutsatsos, I. K., et al. (2017). Jenkins-CI, an Open-Source Continuous Integration System, as a Scientific Data and Image-Processing Platform. SLAS Discovery, 22(3), 238-249. https://doi.org/10.1177/1087057116679993

Siap Melangkah Lebih Jauh?

Jangan biarkan proses rilis manual terus memperlambat tim Anda. Pelajari Docker dan Jenkins lebih dalam melalui *course* Docker Administration atau Linux Practical DevOps, tersedia untuk *engineer* individu maupun program *in-house* yang bisa disesuaikan dengan kebutuhan perusahaan.

Hubungi kami di **training@btech.id** atau kunjungi **https://btech.id/en/trainings/**

professional training