UTS XOps Berdasarkan Alur Proses

IF4054 Software System X-Ops: materi berurutan per tahap,
latihan benar atau salah, dan essay

Daftar Isi

1. Peta Alur dan Cara Belajar **3**

1.1 Satu perubahan menempuh lima tahap 3

1.2 Peta ke catatan sumber 3

1.3 Cara menilai pernyataan benar atau salah 4

2. Fondasi: Mengapa X-Ops **5**

2.1 Materi (baca berurutan) 5

2.2 Latihan: benarkan atau salahkan 8

3. Plan: Merancang Sebelum Membangun **10**

3.1 Materi (baca berurutan) 10

3.2 Latihan: benarkan atau salahkan 13

4. Build: Membangun dan Mengemas **16**

4.1 Materi (baca berurutan) 16

4.2 Latihan: benarkan atau salahkan 19

5. Verify: Memeriksa Sebelum Rilis **22**

5.1 Materi (baca berurutan) 22

5.2 Latihan: benarkan atau salahkan 25

6. Release: Infrastruktur, Gate, dan Rilis **29**

6.1 Materi (baca berurutan) 29

6.2 Latihan: benarkan atau salahkan 32

7. Operate dan Observe: Menjalankan dan Mengawasi **36**

7.1 Materi (baca berurutan) 36

7.2 Latihan: benarkan atau salahkan 38

8. Latihan Campuran: Membenarkan atau Menyalahkan Praktik pada Kasus **41**

9. Essay per Tahap **44**

10. Ringkasan Alur 50

Versi UTS XOps yang disusun mengikuti **alur sebuah perubahan** dari ide sampai berjalan dan diawasi, bukan mengikuti urutan catatan atau minggu kuliah. Isinya tetap slide semester 1 2026/2027 (catatan 17 sampai 26). Versi yang dikelompokkan per catatan ada di UTS XOps, istilah singkat di Glosarium XOps.

Petunjuk

Cara belajar: kerjakan berurutan dari bagian 2 sampai 7, jangan melompat. Di tiap tahap: (1) baca **Materi** dari atas ke bawah, karena setiap langkah menjawab pertanyaan yang ditinggalkan langkah sebelumnya, dan akhir tiap tahap ada **Jembatan** ke tahap berikutnya; (2) kerjakan **Latihan benar atau salah** dengan menutup kolom Kunci; (3) cocokkan alasannya, dan bila salah kembali ke langkah materi yang relevan, bukan menghafal jawabannya. Setelah semua tahap selesai, kerjakan **Latihan campuran** dan **Essay**. Penjelasan yang tidak ada di slide ditandai (tambahan).

1. Peta Alur dan Cara Belajar

1.1 Satu perubahan menempuh lima tahap

Slide W02 Foundations menyebut satu lifecycle: **Plan** → **Build** → **Verify** → **Release** → **Operate**, dengan panah **feedback** dari Operate kembali ke Plan. Setiap tahap dilihat lewat empat lensa: **DevOps** (software), **DataOps** (data), **MLOps** (model), dan **SecOps** (keamanan). Catatan ini mengikuti alur itu, dengan tahap **Fondasi** di awal untuk menjawab pertanyaan "kenapa X-Ops".

Tahap	Pertanyaan utama	Software (DevOps)	Data (DataOps)	Model (MLOps)	Keamanan (SecOps)
Plan	Apa yang dibangun dan bagaimana dirancang?	12-factor, batas microservice per business capability	schema, data contract, batch atau streaming	tujuan bisnis, sumber data	klasifikasi data (PII), least privilege, privacy by design
Build	Bagaimana dibuat dan dikemas?	Git, branching, Dockerfile, image	Kafka (topic, key, partition, offset), bronze	EDA, feature engineering, training	base image minimal, non-root, tanpa secret di kode
Verify	Bagaimana tahu ini layak dirilis?	CI, test pyramid, SAST, SCA, image scan, registry by digest	data quality, compat check schema, GX/dbt, DLQ	evaluation, validation, reproducibility	scan, signing, tokenize dan mask PII
Release	Bagaimana disediakan dan dirilis dengan aman?	Terraform, Kubernetes, canary, GitOps	deploy pipeline ETL	serving, level MLOps	RBAC, Kyverno, secret management
Operate	Bagaimana dijalankan dan diawasi?	monitoring, logging, health check, rollback	consumer lag, DLQ, freshness	model drift, retraining	audit akses, dynamic masking, deteksi intruder

1.2 Peta ke catatan sumber

Tahap	Catatan sumber
Fondasi	17 Course Introduction, 18 XOps Introduction 2026, 19 MLOps DataOps Introduction 2026, 21 Foundations Architecture
Plan	21 Foundations Architecture, 25 DataOps Foundation, 26 DataOps Security Quality Privacy
Build	20 DevOps 2026, 21 Foundations Architecture, 25 DataOps Foundation, 19 MLOps DataOps Introduction 2026
Verify	22 CI and Shift Left, 25 DataOps Foundation, 26 DataOps Security Quality Privacy, 19 MLOps DataOps Introduction 2026
Release	23 IaC and Cluster Security, 24 Release Engineering and Delivery, 19 MLOps DataOps Introduction 2026
Operate	20 DevOps 2026, 25 DataOps Foundation, 26 DataOps Security Quality Privacy, 24 Release Engineering and Delivery, 19 MLOps DataOps Introduction 2026

1.3 Cara menilai pernyataan benar atau salah

Soal pilihan ganda berbentuk pernyataan tentang sebuah praktik, penggunaan, atau penjelasan. Nilai tiap pernyataan sendiri dengan empat pemeriksaan:

- **Istilah atau komponen ditukar.** Contoh: “validating admission controller meng-inject sidecar” salah, karena itu tugas yang mutating.
- **Arah atau urutan terbalik.** Contoh: “shift-left memindahkan pengecekan ke akhir”, “at-most-once = proses dulu baru commit”.
- **Kata mutlak.** “selalu”, “hanya”, “cukup”, “tidak perlu” biasanya membuat pernyataan salah. Pernyataan “boleh asal...” perlu dicek syaratnya.
- **Praktik buruk disajikan sebagai praktik baik.** Contoh: “merge anyway”, tag `latest`, `kubectl apply` untuk hotfix, state Terraform di laptop.

Saat ujian, tulis: **Benar atau Salah**, alasan satu kalimat, dan **perbaikannya** bila salah.

2. Fondasi: Mengapa X-Ops

2.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: masalah → tujuan → cara merilis yang baik (Fowler) → budaya dan praktik (DevOps) → kenapa data dan model butuh perluasan → satu taksonomi (X-Ops). Setiap langkah menjawab pertanyaan yang ditinggalkan langkah sebelumnya, jadi baca dari atas ke bawah.

Langkah 1. Masalahnya: aplikasi bisnis bukan hanya kode

Aplikasi bisnis modern adalah gabungan tiga hal: **data**, **infrastruktur high-available**, dan **cloud on-demand**. Supaya hasilnya reliable, secure, dan produktif, tim development, security, network, dan cloud harus berkolaborasi dengan **IT operations**.

Masalahnya, tim-tim ini sering bekerja terpisah dalam **silo**: developer menulis kode lalu menyerahkannya ke operations. Gerakan **DevOps** (sekitar 2007) lahir untuk meruntuhkan silo itu. Semua materi berikutnya adalah cara menjawab satu pertanyaan: *bagaimana perubahan (code, data, model) sampai ke production dengan cepat, aman, dan bisa diulang?*

Langkah 2. Tujuan: apa yang ingin dicapai X-Ops

Slide menyebut dua goals:

1. Membangun **enterprise technology stack** yang memungkinkan automation dan mengurangi duplikasi.
2. Membuat profesional data dan analytics **meng-operationalize** proses dan otomasinya sejak awal, bukan sebagai afterthought.

Operationalize software artinya pendekatan sistematis untuk mengotomasi build dan mengorkestrasi operasi agar memenuhi tujuan yang terukur dan selaras dengan prioritas bisnis. Kata kuncinya: *sistematis*, *otomasi*, *terukur*.

Pertanyaan berikutnya: seperti apa cara merilis software yang baik itu?

Langkah 3. Cara merilis software yang baik menurut Fowler

(a) Mulai dari yang buruk. Fowler menyebut tiga anti-pattern: (1) deploy manual, (2) production-like environment baru dipakai setelah development selesai, (3) manual configuration management di production. Ketiganya membuat rilis lambat, menegangkan, dan hasilnya tidak bisa diprediksi (penjelasan, tambahan).

(b) Gantinya adalah deployment pipeline. Perubahan melewati tahap berurutan: *commit stage* (compile, unit test, analysis, build installers) → *automated acceptance testing* → *automated capacity testing* → *manual testing* (showcase, exploratory) → *release*. Setiap tahap menyaring perubahan yang belum layak.

(c) Dua kata kunci.

- **Automated.** Software, konfigurasi, environment, dan proses rilis terus berubah. Tanpa otomasi, proses build, deploy, test, dan release **tidak repeatable**: hasilnya berbeda setiap kali dijalankan.
- **Frequent.** Rilis sering berarti **delta** (selisih perubahan) kecil. Delta kecil membuat risiko turun, rollback mudah, dan feedback cepat. Rilis tiga bulan sekali membawa delta besar sehingga bila gagal, sulit melacak penyebabnya.

(d) Alasan terdalam: feedback. Motivasi semuanya adalah mendapat tanggapan atas perubahan. Tiga kriterianya: setiap perubahan memicu feedback, feedback datang secepat mungkin, dan tim menindaklanjutinya. Feedback yang datang lambat atau diabaikan sama saja dengan tidak ada feedback.

(e) Delapan prinsip delivery sebagai rangkuman. Pengelompokan di bawah ini bantuan belajar (tambahan), isinya delapan prinsip slide:

Kelompok	Prinsip	Maknanya
Otomasi dan konsistensi	Repeatable dan reliable release	Rilis bisa diulang dengan hasil sama
	Automate almost everything	Kecuali yang benar-benar butuh penilaian manusia (tambahan)
	Keep everything in version control	Kode, konfigurasi, dan skrip, supaya semua bisa dilacak dan dikembalikan
Frekuensi dan feedback	If it hurts, do it more frequently, and bring the pain forward	Langkah yang menyakitkan dilakukan lebih sering dan lebih awal agar masalah cepat ketahuan
	Build quality in	Kualitas dibangun sejak awal, bukan diperiksa di akhir
Tanggung jawab	Done means released	Selesai bila sudah dirilis, bukan hanya selesai dikoding
	Everybody is responsible	Bukan hanya tim tertentu
	Continuous improvement	Proses terus diperbaiki

(f) Latar: Agile. Agile Manifesto (2001, Snowbird, 17 orang) mengutamakan *individuals and interactions*, *working software*, *customer collaboration*, dan *responding to change*, di atas *process and tools*, *documentation*, *contract*, dan *plan*. Semangat DevOps mirip dengan Agile Manifesto ini.

Langkah 4. DevOps: wadah budaya dan praktiknya

DevOps adalah *practices, tools, dan cultural philosophy* yang mengotomasi dan mengintegrasikan proses antara tim development dan IT; menekankan team empowerment, kolaborasi lintas tim, dan otomasi. Jadi DevOps bukan sekadar tool.

Lifecycle-nya berbentuk **infinity loop**: Discover → Plan → Build → Test → Deploy → Operate → Observe → Continuous feedback, dibungkus Collaboration and Communication. Tafsiran (tambahan): ini adalah prinsip *feedback* Fowler yang dijadikan lingkaran: hasil dari Operate dan Observe kembali menjadi masukan Discover dan Plan berikutnya.

Praktik umum yang mengisi loop itu: **CI**, **CD**, **IaC**, **CaC** (configuration as code), **monitoring dan logging**, serta communication dan collaboration. Setiap praktik ini dibahas di tahap berikutnya.

Langkah 5. Kenapa data dan model memerlukan perluasan

Kode biasa berubah hanya saat kita mengubahnya. **Software berbasis ML** berbeda, dan perbedaan inilah yang membuat DevOps saja tidak cukup:

Karakteristik	Mengapa menambah beban operasional
Data dependency	Perilaku model ditentukan data, bukan hanya kode
Model drift	Performa turun karena data nyata berubah, sehingga perlu monitoring dan retraining
Proses eksperimental	Perlu model version control dan tracking untuk melacak hasil tiap percobaan

Karakteristik	Mengapa menambah beban operasional
Specialized tools	Butuh tool khusus di luar tool software biasa
Integrasi dengan tim data science	Dua budaya kerja harus disatukan
Continuous evaluation	Performa harus terus dievaluasi lewat feedback loop, bukan sekali di akhir

Dua disiplin menjawab ini:

- **MLOps** = DevOps untuk ML engineering: developing, testing, deploying, monitoring, dan operating sistem ML. Letaknya di irisan **Machine Learning, Data Engineering, dan DevOps**.
- **DataOps** = practices, procedures, dan technologies data management yang *integrated dan process-oriented* untuk meningkatkan quality, speed, dan collaboration, dengan hasil data products dan data apps. Key roles: developers, data engineers, analysts, data scientist, dan business users.

Hubungannya: DataOps adalah fondasi (*data as a foundation*) karena model hanya sebaik data yang masuk; keduanya berbagi tujuan (automation, collaboration, continuous improvement) dalam satu integrated workflow.

Langkah 6. Menyatukan semuanya: taksonomi X-Ops

XOps berarti menerapkan disiplin yang sama (**automation, versioning, testing, monitoring**) ke setiap artifact: code, data, dan models. "X" diganti sesuai artifact-nya. Taksonomi ini untuk **shared vocabulary**, bukan tembok antar-disiplin.

Pilar	Fokus
DevOps	Build, test, release software reliable dan frequent; fondasi pilar lain. Diukur dengan delivery metrics: deployment frequency, lead time, change failure rate, time to restore
DataOps	CI/CD untuk data pipeline: ingestion, quality, schema, lineage. Quality checks menjadi pipeline gate
MLOps	DevOps untuk model lifecycle; model diperlakukan sebagai versioned artifact. Continuous Training (CT) dipicu data baru atau performance decay
SecOps	Lensa di setiap tahap, bukan ditempel di akhir. Shift-left security: scanning, least privilege, policy saat build time, bukan audit time

Cara membaca tabel: tiga pilar pertama menjawab *artifact apa yang dikelola*; pilar keempat menjawab *kapan keamanan diterapkan*, yaitu di semua tahap sejak awal.

Langkah 7. Info mata kuliah (singkat)

- IF4054 Software System X-Ops, 3 SKS, dosen Yudistira Asnar dan Bayu Samudra. Penilaian: assignments 2 sampai 3 kali, quiz, mid, final. Minimum kehadiran 70%. Kecurangan dihukum berat: memalsukan kehadiran, menyalin pekerjaan orang lain, meminta GPT atau AI sejenis menulis kode proyek.
- Enam outcome: memahami prinsip dan praktik X-Ops; hands-on containerization dan orchestration; mengimplementasikan CI/CD; mengintegrasikan dan mengotomasi tahap software dan data lifecycle; menjalankan praktik X-Ops; merancang sistem software untuk X-Ops.

Jembatan ke tahap Plan

Kita sudah tahu *kenapa* perlu X-Ops dan *apa* yang diotomasi. Pertanyaan berikutnya: **supaya semua otomasi itu bisa jalan, sistem harus dirancang seperti apa?** Rancangan yang buruk (config di kode, state di memori, schema berubah sembarangan) tidak bisa diotomasi dengan aman. Itu isi tahap Plan.

2.2 Latihan: benarkan atau salahkan

Latihan F1: Praktik rilis dan prinsip delivery

#	Pernyataan	Kunci	Alasan
1	Tim merilis tiga bulan sekali agar risikonya terkendali karena jarang.	**Salah**	Rilis jarang membuat delta besar. Fowler: **frequent** membuat delta kecil, risiko turun, rollback mudah, feedback cepat.
2	Deploy dengan menyalin file dan mengubah konfigurasi production secara manual adalah praktik yang wajar.	**Salah**	Itu dua anti-pattern Fowler (deploy manual, manual configuration management production). Perbaikan: otomasi dan version control.
3	Proses build, deploy, test, dan release yang tidak diotomasi tidak repeatable.	**Benar**	Hasilnya berbeda tiap dijalankan karena software, konfigurasi, environment, dan proses rilis berubah.
4	"Done means released": fitur selesai bila sudah dirilis, bukan hanya selesai dikoding.	**Benar**	Salah satu prinsip delivery Fowler.
5	"If it hurts, do it more frequently" berarti langkah yang menyakitkan sebaiknya ditunda sampai akhir proyek.	**Salah**	Justru dilakukan lebih sering dan lebih awal (bring the pain forward) supaya masalah cepat ketahuan.

Latihan F2: DevOps dan XOps

#	Pernyataan	Kunci	Alasan
1	DevOps pada dasarnya adalah sebuah tool otomasi.	**Salah**	DevOps = practices, tools, dan cultural philosophy.
2	Gerakan DevOps muncul karena developer dan operations bekerja terpisah dalam silo.	**Benar**	Dimulai sekitar 2007; "no silo" adalah semangatnya.
3	Lifecycle DevOps berakhir di tahap Operate.	**Salah**	Infinity loop: Operate → Observe → Continuous feedback, lalu kembali ke siklus berikutnya.
4	XOps berarti menerapkan automation, versioning, testing, dan monitoring yang sama ke code, data, dan models.	**Benar**	"X" berganti sesuai artifact: DevOps, DataOps, MLOps, SecOps.
5	SecOps adalah tahap terpisah yang dijalankan di akhir sebelum release.	**Salah**	SecOps adalah lensa di setiap tahap; shift-left security dilakukan saat build time, bukan audit time.

Latihan F3: ML, MLOps, DataOps

#	Pernyataan	Kunci	Alasan
1	Model drift berarti performa model bisa turun karena data nyata berubah, sehingga perlu monitoring dan retraining.	**Benar**	Salah satu karakteristik software ML.
2	Performa model cukup dievaluasi sekali saat training.	**Salah**	Perlu continuous evaluation dengan feedback loop.
3	MLOps berada di irisan Machine Learning, Data Engineering, dan DevOps.	**Benar**	Diagram Venn di slide W01b.
4	DataOps hanya urusan tim data engineer.	**Salah**	Key roles: developers, data engineers, analysts, data scientist, business users.
5	DataOps menjadi fondasi MLOps karena menjamin suplai data yang andal dan konsisten.	**Benar**	“Data as a foundation”.

3. Plan: Merancang Sebelum Membangun

3.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: pisahkan yang berubah dari yang bervariasi (12-factor) → pecah aplikasi di batas yang benar (microservices) → pilih cara data mengalir (batch atau streaming) → sepakati kontrak data → rancang keamanan dan tata kelola dari awal. Prinsip yang sama dipakai untuk code dan untuk data.

Langkah 1. Pisahkan yang berubah dari yang bervariasi: 12-Factor App

Masalah. Aplikasi "jalan di laptop developer, rusak di production". Sebabnya hampir selalu sama: kode bercampur dengan hal yang berbeda antar-environment (alamat database, password, jumlah resource).

Ide. 12-Factor App adalah metodologi dari engineer Heroku setelah mengamati ratusan aplikasi. Satu ide dasarnya: pisahkan **yang berubah (code)** dari **yang bervariasi (config, environment)**. Dua belas faktornya bisa dibaca dalam tiga kelompok (pengelompokan ini bantuan belajar, tambahan):

Kelompok 1: kode, dependency, dan config dipisah (I sampai IV)

No	Faktor	Inti	Kenapa penting
I	Codebase	Satu codebase di version control, banyak deploy; jangan fork per environment	Fork per environment membuat kode menyimpang dan sulit disatukan
II	Dependencies	Deklarasikan dan isolasi dependency secara eksplisit	Tidak bergantung pada paket yang kebetulan terpasang di mesin
III	Config	Config (credentials, endpoints, feature flags) di environment variables, bukan di kode	Kode yang sama bisa jalan di semua environment; secret tidak bocor ke Git
IV	Backing services	Database, queue, cache sebagai attached resources, ditukar lewat config	Mengganti database lokal ke cloud cukup ubah config

Kelompok 2: bagaimana aplikasi dibangun dan dijalankan (V sampai IX)

No	Faktor	Inti	Kenapa penting
V	Build, release, run	Pisahkan tegas build, release (build + config), dan run	Hasil build yang sama bisa dirilis ke environment mana pun
VI	Processes	Stateless process; data persisten di backing service	Proses boleh mati kapan saja tanpa kehilangan data
VII	Port binding	Self-contained, ekspos service lewat port	Tidak bergantung pada web server eksternal
VIII	Concurrency	Scale out dengan menambah process	Kapasitas ditambah dengan menambah instance, bukan membesarkan satu proses
IX	Disposability	Fast startup dan graceful shutdown	Orchestrator bisa kill dan reschedule dengan cepat

Kelompok 3: konsistensi dan pengamatan (X sampai XII)

No	Faktor	Inti	Kenapa penting
X	Dev/prod parity	Development, staging, production semirip mungkin	Bug tidak baru muncul di production
XI	Logs	Log sebagai event stream ke stdout	Routing dan penyimpanan log diurus environment, bukan aplikasi
XII	Admin processes	Task admin sebagai one-off process di environment identik	Migrasi atau skrip admin berperilaku sama seperti aplikasi

Kaitan dengan container. Image hampir by definition adalah 12-factor artifact (tanpa config yang di-bake). Factor III dipetakan ke `--env` di Docker dan ConfigMaps dan Secrets di Kubernetes; Factor IX memungkinkan orchestrator kill dan reschedule container. **Batas "build once, run everywhere":** arsitektur ARM vs x86 (butuh multi-arch image), workload yang bergantung hardware (GPU), backing service tetap berbeda antar environment, dan race condition yang baru muncul di skala production.

Langkah 2. Pecah aplikasi di batas yang benar: microservices

Ide. Microservices memecah aplikasi menjadi service yang di-deploy independen, masing-masing memegang satu **business capability** (mis. orders, inventory, billing). Untungnya: independent scaling, deployment, dan failure isolation. Biayanya: **network call menggantikan function call**, jadi lebih lambat dan bisa gagal.

Kenapa per business capability, bukan per technical layer. Bila dipecah per layer (UI, logic, data), hampir setiap aksi user harus melewati beberapa service sekaligus. Batas yang buruk menghasilkan **distributed monolith**: terlihat terpisah, tetapi tetap saling terkunci (penjelasan, tambahan).

Cara service berinteraksi:

- **API Gateway**: single entry point antara client dan service, mengurus routing dan auth.
- **State**: service stateless (Factor VI); tiap service punya **data store sendiri** (*database-per-service*). Mengakses database service lain merusak encapsulation, sehingga perubahan internal satu service bisa merusak yang lain. Konsistensi antar-service (saga, eventual consistency) dibahas di DataOps.

Batas lain yang perlu diingat. Config separation mengatasi "works on my machine" dan secret di kode, tetapi **tidak** mengatasi **blast radius** (cakupan dampak bila gagal). Menurut **Conway's Law**, sistem mencerminkan struktur komunikasi organisasi; tim yang kusut menghasilkan distributed monolith. Artinya batas teknis dan batas tim harus dirancang bersama (simpulan, tambahan).

Langkah 3. Data: pilih cara mengalir dan sepakati kontraknya

(a) Kenapa data lebih sulit daripada code. Data berubah setiap kali dunia berubah, bukan hanya saat kita deploy. Bug data sering gagal diam-diam (tidak ada error, tetapi hasil salah). Input tidak terbatas dan berantakan. Rollback berarti menulis ulang histori. Producer dan consumer dipegang tim berbeda. Karena itu prinsip rancangan di atas harus diperluas ke data.

(b) Pilih paradigma: batch atau streaming.

	Batch	Streaming
Data	Bounded, diproses setelah lengkap	Unbounded, diproses saat event datang
Latency	Menit sampai jam	Milidetik sampai detik
Complexity	Lebih rendah	Lebih tinggi (state, ordering, retries)

	Batch	Streaming
Cost model	Burst compute	Always-on
Reprocessing	Jalankan ulang job	Butuh replayable log
Failure mode	Job terlambat	Lag, duplikat, poison message

Cara memilih: tanyakan **seberapa cepat keputusan dibutuhkan**. Laporan revenue harian nightly → batch (freshness tidak kritis). Fraud check saat checkout → streaming (keputusan dalam hitungan detik).

(c) Pilih cara komunikasi. Point-to-point (REST, gRPC): tight coupling, gagal bila consumer mati. **Pub/Sub** (Kafka, RabbitMQ): producer publish ke topic lewat broker, loose coupling (decoupled dalam time, space, rate), pesan di-buffer saat consumer mati. Pada streaming, pub/sub-lah yang memungkinkan producer tidak perlu tahu siapa consumer-nya.

(d) Sepakati kontrak data. Schema adalah API: producer dan consumer dipegang tim berbeda, jadi perubahan harus eksplisit dan bisa dicek; tanpa itu, satu rename di producer bisa merusak semua consumer. **Schema-on-write** (database) kaku tetapi mencegah data buruk; **schema-on-read** (data lake) fleksibel tetapi berisiko garbage in garbage out.

Format	Ukuran	Schema ditegakkan	Evolusi	Cocok untuk
JSON	Verbose	Tidak	Ad hoc	API, debugging
Avro	Compact	Ya	Schema resolution	Data pipeline
Protobuf	Compact	Ya	Field numbers	Service-to-service

Schema akan berubah. Aturan **compatibility** menentukan perubahan mana yang aman:

- **Backward**: schema baru bisa membaca data lama. **Forward**: schema lama bisa membaca data baru. **Full**: keduanya.
- **Safe**: menambah field optional yang punya default. **Breaking**: menghapus atau me-rename field required, mengubah tipe.

Data contract lebih luas dari schema: **schema + semantics + ownership** (nama, makna, dan tipe field; required vs optional; rentang nilai; delivery guarantee dan freshness; klasifikasi PII per field). Data contract dicek di sisi producer sebelum publish. Isinya bisa dilihat dalam lima lapisan; lapisan semantics dan SLA ditegakkan dengan Great Expectations, dbt, dan freshness check, yang berjalan setelah data ada:

Lapisan contract	Contoh	Ditegakkan oleh
Structure	`currency_code` adalah string	Avro atau Protobuf saat serialization
Domain	`currency_code` ada di ISO 4217	Avro enum atau expectation Great Expectations
Semantics	`amount` tidak negatif, `transaction_type` ada di charge atau refund	Great Expectations atau dbt tests
SLA	event tiba dalam 5 menit dari timestamp	freshness check dan alert
Ownership	siapa yang di-page bila gagal	contract metadata

Langkah 4. Rancang tata kelola dan keamanan dari awal

Prinsipnya: kita tidak bisa melindungi apa yang tidak bisa diidentifikasi. Karena itu klasifikasi data harus ada sebelum kontrol diterapkan.

(a) Kerangka: DAMA-DMBOK. Kosakata dan framework untuk mengelola data sebagai aset:

11 knowledge area dengan **Data Governance** di tengah. Ia **bukan tool atau metodologi**; ia menyebut area apa yang harus dicakup, bukan produk apa yang dibeli. Tiga area dipakai sebagai lensa (Data Quality, Data Security, Metadata Management), diikat Data Governance.

(b) Dari metadata ke kontrol otomatis. Alurnya: data catalog (di mana field berada) → business glossary (arti field) → classification tags (`PII`, `PCI`, `Confidential`, `Public`) → policy engine (PII → tokenize, PCI → mask, public → simpan apa adanya). Metadata menggerakkan otomasi: kontrol diterapkan berdasarkan tag, bukan tebakan manual.

(c) Security dan privacy itu berbeda. Security melindungi data dari akses, perubahan, dan kehilangan tidak sah (**CIA triad**). **Privacy** menyangkut hak orang yang datanya dibicarakan (purpose, consent, minimization, retention). Security tanpa privacy mungkin; privacy tanpa security tidak, sebab janji privacy tidak bisa ditegakkan tanpa kontrol akses. **PCI-DSS** adalah standar security industri; **GDPR dan UU PDP** adalah undang-undang privacy.

(d) Jebakan umum. Pseudonymized data tetap data pribadi menurut GDPR: tokenization mengurangi exposure, bukan kewajiban hukum.

(e) Privacy by design (contoh PayLend). `ssn` → tokenize; kartu → mask; `home_address` → delete karena tidak dibutuhkan (minimisasi). Aturan praktis: data yang tidak dikumpulkan tidak bisa bocor.

Langkah 5. Apa yang dikerjakan di tahap Plan

- **DataOps (Planning):** define goals and objectives, identify data sources, establish data quality standards.
- **Model:** establish business objectives dan tentukan data sources.

Jembatan ke tahap Build

Rancangan sudah ada: aplikasi stateless dengan config terpisah, batas service jelas, cara data mengalir dipilih, kontrak dan klasifikasi disepakati. Pertanyaan berikutnya: **bagaimana rancangan itu diwujudkan dan dikemas supaya berjalan sama di laptop, CI, dan production?** Itu isi tahap Build.

3.2 Latihan: benarkan atau salahkan

Latihan P1: 12-Factor App

#	Pernyataan	Kunci	Alasan
1	Menyimpan password database di environment variable memenuhi Factor III (Config).	**Benar**	Config (credentials, endpoints, feature flags) di environment variables, bukan di kode.
2	Menyimpan isi keranjang belanja di memori proses agar cepat sesuai Factor VI (Processes).	**Salah**	Proses harus stateless; data persisten di backing service.
3	Membuat fork codebase terpisah untuk tiap environment diperbolehkan.	**Salah**	Factor I: satu codebase di version control, banyak deploy.
4	Menulis log ke stdout dan membiarkan execution environment yang mengurus routing dan storage memenuhi Factor XI.	**Benar**	Log diperlakukan sebagai event stream.
5	Scaling mengikuti Factor VIII dilakukan dengan membesarkan satu proses.	**Salah**	Scale out dengan menambah instance stateless process.

Latihan P2: Microservices dan arsitektur

#	Pernyataan	Kunci	Alasan
1	Memecah service per technical layer (UI, logic, data) adalah cara yang direkomendasikan.	**Salah**	Pecah per business capability; batas per layer memaksa cross-service call.
2	Service orders boleh membaca langsung tabel milik service inventory bila lebih cepat.	**Salah**	Database-per-service; akses langsung merusak encapsulation.
3	API Gateway berperan sebagai single entry point, routing, dan auth bagi client.	**Benar**	Antara client dan service seperti Orders dan Inventory.
4	Config separation sudah cukup untuk mencegah kesalahan satu tim menjadi incident semua orang.	**Salah**	Tidak mengatasi blast radius; menurut Conway's Law tim yang kusut menghasilkan distributed monolith.
5	Microservices menukar satu deployment besar dengan banyak deployment kecil, dengan biaya network call menggantikan function call.	**Benar**	Untung independent scaling, deployment, failure isolation.

Latihan P3: Desain data

#	Pernyataan	Kunci	Alasan
1	Fraud check saat checkout cocok memakai nightly batch karena freshness tidak kritis.	**Salah**	Keputusan dibutuhkan dalam hitungan detik, jadi streaming. Nightly batch cocok untuk laporan revenue harian.
2	Karena producer dan consumer dipegang tim berbeda, schema berperan seperti API dan perubahannya harus eksplisit dan bisa dicek.	**Benar**	Tanpa contract, satu rename di producer bisa merusak semua consumer.
3	Data contract hanyalah schema (nama dan tipe field).	**Salah**	Data contract = schema + semantics + ownership.
4	Pada Pub/Sub, producer harus tahu alamat setiap consumer.	**Salah**	Producer dan consumer decoupled dalam time, space, rate; producer tidak tahu siapa consumer-nya.
5	Menambah field optional dengan default adalah perubahan schema yang aman.	**Benar**	Backward compatible; yang breaking adalah hapus atau rename field required dan ubah tipe.

Latihan P4: Keamanan dan tata kelola saat merancang

#	Pernyataan	Kunci	Alasan
1	DMBOK adalah tool yang harus dibeli untuk mengelola data.	**Salah**	DMBOK adalah framework dan kosakata (11 knowledge area), bukan tool atau metodologi.
2	Field diklasifikasi (PII, PCI, public) lewat catalog dan classification tags karena data yang tidak bisa diidentifikasi tidak bisa dilindungi.	**Benar**	Metadata menggerakkan otomasi: policy engine menerapkan kontrol berdasarkan tag.

#	Pernyataan	Kunci	Alasan
3	Security tanpa privacy mungkin terjadi, tetapi privacy tanpa security tidak.	**Benar**	Privacy tanpa security hanya janji yang tidak bisa ditegakkan.
4	PCI-DSS adalah undang-undang privacy.	**Salah**	PCI-DSS standar security industri; GDPR dan UU PDP undang-undang privacy.
5	Data yang sudah di-tokenize bebas dari kewajiban privacy.	**Salah**	Pseudonymized data tetap data pribadi; tokenization mengurangi exposure, bukan obligations.

4. Build: Membangun dan Mengemas

4.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: kode disimpan dan dikelola (Git) → kenapa perlu container → bagaimana container dibuat dan dijalankan → aplikasi mana yang cocok → bagaimana mengamankannya sejak build → lalu hal yang sama untuk jalur data (Kafka) dan model.

Langkah 1. Simpan dan kelola kode: Git dan branching

Kenapa Git. Git adalah distributed version control: setiap clone adalah salinan penuh history, jadi tidak ada single point of failure. Commit membentuk directed graph; **branch hanyalah pointer ringan**, jadi murah dibuat. Git memenuhi **Factor I** (satu codebase di version control) dan prinsip "keep everything in version control".

Aturan kerja (branching strategy kuliah):

1. **main** selalu **deployable**: apa pun yang ada di main siap dirilis.
2. Pekerjaan dilakukan di branch `feature/*` yang **short-lived**, satu milestone saja. Branch panjang menumpuk perbedaan dan menyulitkan penggabungan (*merge hell*).
3. Perubahan masuk lewat **Pull Request** (review + CI).
4. Digabung dengan **squash**: semua commit branch menjadi satu commit di main, supaya history main rapi (tambahan).

Langkah 2. Kenapa container: dari "works on my machine" ke "package once, deploy anywhere"

Masalah. Menjalankan software terdistribusi yang andal di environment komputasi apa pun itu sulit (penyebab umum: perbedaan OS, library, dan dependency; tambahan). **Ide container:** kemas aplikasi beserta dependency-nya sehingga bisa berjalan sama di mana saja: **package once, deploy anywhere**. Namanya dari shipping container: standar sehingga barang apa pun bisa diangkut tanpa loading dan unloading berulang.

Bagaimana container berbeda dari server fisik dan VM. Bandingkan susunan layer-nya:

Model	Susunan layer	Catatan
Physical server	Hardware → OS → libraries dan dependencies bersama → apps	Dependency mudah bentrok antar-aplikasi
Virtual machine	Hardware → Host OS → Hypervisor → tiap VM punya Guest OS + libraries → app	Isolasi penuh, tetapi tiap VM membawa OS lengkap (berat)
Container	Hardware → OS → Container Runtime → tiap container hanya libraries dan dependencies + app	Tanpa Guest OS; terisolasi dan ringan

Aspek	Container	Virtual machine
Isolasi	Berbagi kernel OS host, isolasi level proses	Memvirtualisasi hardware, guest OS lengkap
Ukuran	Megabyte per image	Resource footprint lebih berat
Waktu start	Milidetik sampai beberapa detik	Puluhan detik sampai menit

Aspek	Container	Virtual machine
Paling cocok untuk	Microservice (banyak, kecil, disposable)	Saat benar-benar butuh kernel atau OS berbeda

Inilah yang membuat container cocok untuk microservice dan Factor IX (disposability): start cepat dan murah dibuang.

Langkah 3. Cara container dibuat dan dijalankan

Alur kerja: tulis kode → buat **Dockerfile** → build, run, dan unit test image → push image ke **container registry** (mis. Docker Hub) → jalankan container dari image di cluster.

Image dan container. Image adalah artifact **immutable** dan berlapis; container adalah running instance dengan writable layer tipis. Urutan layer: Base Image → Dependencies → App Code → Entrypoint.

Kenapa urutan layer penting. Layer yang tidak berubah diambil dari build cache. Karena itu `COPY requirements.txt` dan install dependency dilakukan **sebelum** `COPY . .`: kode lebih sering berubah daripada dependency, jadi perubahan kode tidak memicu install ulang semua package (penjelasan cache: tambahan).

Multi-stage build memisahkan build environment dari runtime environment, sehingga image akhir hanya berisi komponen esensial dan tools build tidak ikut ke production.

"Works on my machine" diselesaikan karena Dockerfile ada di version control: image identik dan reproducible di lokal, CI, dan production.

Docker Compose dan MinIO. Docker Compose menyalakan service beserta backing service-nya (database, MinIO) dalam satu file sebagai attached resources (**Factor IV**); environment lokal identik untuk tiap anggota tim. **MinIO** adalah object storage S3-compatible: offline, gratis, bisa di-reset bersih, dan kode yang sama diarahkan ke cloud storage di production lewat config (Factor III).

Langkah 4. Tidak semua aplikasi cocok di-containerize

Aplikasi yang cocok: **stateless**; **high cohesion** (microservice, API-first); sering di-update; kompatibel dengan teknologi container; dependency dan konfigurasi bisa didefinisikan dari luar; **tidak** dijaga specialized team atau di-maintain banyak tim; direncanakan jalan di environment homogen. Sebagian kriteria ini sejalan dengan 12-factor di tahap Plan (stateless, config dari luar); sisanya soal praktik organisasi dan lingkungan.

Migrasi VM → container berjalan bertahap: Discovery → Application Requirement Assessment → Container Infrastructure Design → Containerize Application → Testing → Deployment and Rollout, lalu bisa kembali ke Discovery untuk bagian berikutnya.

Langkah 5. Keamanan sejak build

Container yang mudah dibuat juga mudah dibuat dengan buruk. Praktik yang diterapkan sejak build:

Praktik	Alasan
Minimal base image	Mengurangi attack surface: tidak ada paket yang tidak perlu
Least privilege: jangan jalankan sebagai root	Container yang dibobol tidak langsung berkuasa
Immutable infrastructure: image tidak diubah setelah di-build dan di-deploy; ada perubahan → build image baru	Mencegah perbedaan tersembunyi antar-instance

Praktik	Alasan
Digital signature pada image	Memverifikasi authenticity dan integrity
Secure registry dengan access control	Hanya pihak berwenang yang boleh push dan pull
TLS antar-container, network segmentation, update dan patch rutin	Membatasi penyebaran serangan
Credential tidak pernah di kode; lewat environment variable atau secret	Sejalan dengan Factor III
Praktik performa: resource requests dan limits, caching, health check, orchestration	Menjaga container tetap efisien dan hanya yang sehat melayani traffic

Langkah 6. Membangun jalur data: Kafka

Bagian ini mengikuti urutan pertanyaan yang muncul saat membangun pipeline streaming. Contoh konkret di slide (ShopStream): topic `shopstream.orders.v1`, dead-letter topic `shopstream.orders.dlq`, key `order_id`.

(a) Apa itu Kafka? Sebuah **log** yang append-only, ordered, dan immutable. Tiap record punya **offset**; consumer mencatat posisinya sendiri sehingga log bisa di-**replay**. Karena itu Kafka bukan sekadar queue yang menghapus pesan setelah dibaca.

(b) Bagaimana skala dan urutan dijaga? Topic dipecah menjadi **partition** untuk parallel processing dan horizontal scaling. Pesan masuk partition berdasarkan **key** (di-hash). **Ordering hanya dijamin di dalam satu partition.** Konsekuensinya: event yang urutannya penting harus punya key yang sama (mis. `order_id` untuk semua event satu order).

(c) Siapa yang membaca? Consumer group: setiap partition dibaca tepat satu consumer dalam group; menambah consumer untuk scaling maksimal sebanyak jumlah partition (consumer lebihnya menganggur, tambahan); group berbeda membaca independen. **Committed offset** artinya "saya sudah memproses sampai sini". **Retention** membatasi seberapa jauh bisa replay; **compaction** hanya menyimpan nilai terbaru per key.

(d) Apa yang terjadi bila consumer crash? Di sinilah **delivery semantics** muncul. Semuanya bergantung pada urutan antara *memproses* dan *commit offset*:

Guarantee	Urutan	Risiko
At-most-once	Commit dulu, baru proses	Data loss
At-least-once	Proses dulu, baru commit	Duplikat
Exactly-once	Setiap efek terjadi sekali	Sulit, cakupan terbatas

Aturan praktis: commit **setelah** side effect berhasil. **Duplikat muncul** bila consumer crash setelah menulis ke storage tetapi sebelum commit offset: saat restart, batch yang sama diproses ulang.

(e) Bagaimana mengatasi duplikat? Effectively-once = at-least-once + idempotency. Consumer dibuat idempotent: dedupe berdasarkan `event_id`, atau pakai nama object deterministik `topic-partition-startOffset` sehingga penulisan ulang hanya menimpa. Producer juga diatur `enable.idempotence=true` dan `acks=all`.

(f) Ke mana data ditulis? Pola **medallion**: **bronze** (raw, immutable, jangan pernah diubah) → **silver** (cleaned, masked) → **gold** (features, marts). Alasan menyimpan raw dulu: reprocessing, auditing, schema-on-read. **Layout lake:** partisi `dt=/hour=` berdasarkan event time atau ingest time untuk pruning dan backfill; hindari **small-file problem** (satu file kecil per event) dengan buffer berdasarkan ukuran atau waktu.

(g) Bagaimana schema dikelola? Schema Registry: producer register atau fetch schema id,

mengirim **id + payload** ke Kafka; consumer fetch schema berdasarkan id lalu men-deserialize. **Avro**: schema JSON (`.avsc`), payload compact, evolusi bawaan. Menambah field aman dengan union `["null","string"]` default `null`.

(h) Task DataOps di tahap Development: data extraction, data transformation (clean, normalize), data loading ke data warehouse atau data lake.

Langkah 7. Membangun model

Tujuh langkah model development, berurutan: (1) establish business objectives → (2) data sources → (3) **EDA** (exploratory data analysis, memahami data sebelum memodelkan) → (4) feature engineering and selection → (5) training and evaluation → (6) validation → (7) **reproducibility** (hasil bisa diulang dengan data, kode, dan parameter yang sama). Karena prosesnya eksperimental, eksperimen dilacak lewat **model version control dan tracking**.

Jembatan ke tahap Verify

Kode, image, dan consumer data sudah ada. Pertanyaan berikutnya: **bagaimana kita tahu semua itu layak dirilis?** Memeriksa di akhir mahal; kita ingin memeriksa otomatis, sedini dan semurah mungkin. Itu isi tahap Verify.

4.2 Latihan: benarkan atau salahkan

Latihan B1: Git dan alur kerja

#	Pernyataan	Kunci	Alasan
1	Branch main harus selalu deployable.	**Benar**	Aturan branching strategy kuliah.
2	Branch `feature/*` sebaiknya long-lived sampai semua fitur selesai.	**Salah**	Short-lived, satu milestone; branch panjang membawa merge hell.
3	Setiap clone Git adalah salinan penuh history, sehingga tidak ada single point of failure.	**Benar**	Berbeda dari centralized VCS.
4	Branch di Git adalah salinan seluruh repository sehingga mahal dibuat.	**Salah**	Branch hanyalah pointer ringan di dalam commit graph.
5	Merge dengan squash menggabungkan semua commit branch menjadi satu commit di main.	**Benar**	Aturan merge di branching strategy kuliah.

Latihan B2: Container dan image

#	Pernyataan	Kunci	Alasan
1	Setiap container membawa Guest OS sendiri.	**Salah**	Container tanpa Guest OS; yang membawa Guest OS adalah VM.
2	Container berbagi kernel host dengan isolasi level proses dan start dalam milidetik sampai detik.	**Benar**	VM butuh puluhan detik sampai menit.
3	Meng-copy `requirements.txt` dan menginstal dependency sebelum `COPY . .` menjaga layer dependency tetap ter-cache saat kode berubah.	**Benar**	Perubahan kode tidak memicu install ulang semua package.
4	Multi-stage build memisahkan build environment dari runtime environment sehingga image akhir hanya berisi komponen esensial.	**Benar**	Tools build tidak ikut terbawa ke production.

#	Pernyataan	Kunci	Alasan
5	Aplikasi yang menyimpan state di memori dan dijaga satu tim khusus adalah kandidat yang baik untuk di-containerize.	**Salah**	Kriteria: stateless, dan tidak dijaga specialized team.

Latihan B3: Keamanan container dan migrasi

#	Pernyataan	Kunci	Alasan
1	Menjalankan container sebagai root agar tidak terkendala permission.	**Salah**	Least privilege: hindari root dan batasi akses.
2	Image yang sudah berjalan boleh di-patch langsung di dalam container.	**Salah**	Immutable infrastructure: build image baru.
3	Minimal base image mengurangi attack surface.	**Benar**	Hindari package dan dependency yang tidak perlu.
4	Image ditandatangani dengan digital signature untuk memverifikasi authenticity dan integrity.	**Benar**	Benar dari pembuatnya dan tidak diubah.
5	Urutan migrasi VM ke container: discovery, requirement assessment, container infrastructure design, containerize, testing, deployment dan rollout.	**Benar**	Siklus dan bisa kembali ke Discovery.

Latihan B4: Kafka

#	Pernyataan	Kunci	Alasan
1	Ordering dijamin di seluruh partition dalam satu topic.	**Salah**	Ordering hanya dijamin di dalam satu partition.
2	Event dengan key yang sama (mis. `order_id`) masuk ke partition yang sama sehingga urutannya terjaga.	**Benar**	Key di-hash untuk menentukan partition.
3	Dalam satu consumer group, satu partition dibaca tepat satu consumer.	**Benar**	Group berbeda membaca secara independen.
4	Commit offset sebelum menulis ke storage menghilangkan risiko kehilangan data.	**Salah**	Itu at-most-once: tidak ada duplikat, tetapi bila crash sebelum menulis, data hilang. Commit setelah side effect berhasil.
5	Log Kafka append-only dan immutable sehingga bisa di-replay; inilah yang membedakannya dari queue biasa.	**Benar**	Consumer mencatat offset-nya sendiri.

Latihan B5: Delivery semantics dan penyimpanan

#	Pernyataan	Kunci	Alasan
1	At-most-once: commit dulu baru proses, risikonya data loss.	**Benar**	At-least-once kebalikannya, risikonya duplikat.
2	Consumer crash setelah menulis ke MinIO tetapi sebelum commit menghasilkan duplikat saat restart.	**Benar**	Batch yang sama diproses dan ditulis ulang.

#	Pernyataan	Kunci	Alasan
3	Effectively-once dicapai dengan at-most-once ditambah retry.	**Salah**	Effectively-once = at-least-once + idempotency.
4	Menyimpan satu file kecil per event ke data lake adalah praktik yang baik.	**Salah**	Small-file problem; buffer berdasarkan ukuran atau waktu.
5	Layer bronze boleh diubah untuk membersihkan data.	**Salah**	Bronze tidak pernah diubah; silver dan gold diturunkan darinya.

Latihan B6: Pengembangan model dan jalur data

#	Pernyataan	Kunci	Alasan
1	Eksperimen model perlu model version control dan tracking karena prosesnya eksperimental.	**Benar**	Mencatat versi model dibuat dari data, kode, dan parameter apa.
2	Reproducibility berarti hasil model bisa diulang dengan data, kode, dan parameter yang sama.	**Benar**	Langkah ke-7 model development.
3	Feature engineering dilakukan sebelum EDA.	**Salah**	EDA (langkah 3) mendahului feature engineering and selection (langkah 4).
4	Reproducibility bukan bagian dari langkah model development.	**Salah**	Reproducibility adalah langkah terakhir dari tujuh langkah.
5	Tahap Development di DataOps berisi extraction, transformation, dan loading data.	**Benar**	Loading ke data warehouse atau data lake.

5. Verify: Memeriksa Sebelum Rilis

5.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: kenapa bug harus ditemukan lebih awal (shift-left) → prinsip CI → isi pipeline build-time (test, SAST, SCA, image scan) → bagaimana artifact dijaga dari tampering (registry, supply chain) → lalu logika yang sama untuk data (quality, contract, PII) dan model.

Langkah 1. Kenapa bug harus ditemukan sedini mungkin

Cost-of-defects curve: bug yang ditemukan di IDE atau pre-commit biayanya **1x**, setelah merge dan ketahuan di CI **15x**, setelah sampai production **100x+**. Makin lambat ditemukan, makin banyak orang, kode, dan data yang sudah terdampak. **Shift-left** artinya memindahkan pengecekan ke tahap yang lebih awal (kiri di garis waktu), tempat perbaikan paling murah.

Tahap	Pengecekan
Code	IDE linting, pre-commit hooks
Commit	Unit test, SAST pada diff
Build	Compile, dependency scan
Package	Container scan, sign artifact
Deploy	Policy checks (last line of defense)

Pengecekan tidak digantikan oleh tahap berikutnya; tiap tahap menangkap jenis masalah yang berbeda, dan deploy adalah pertahanan terakhir, bukan satu-satunya.

Langkah 2. Continuous Integration dan empat prinsipnya

CI: merge kode ke shared trunk **beberapa kali sehari**; tiap merge memicu build dan test otomatis. Tujuannya membuat integrasi jadi *non-event*. CI terus menjawab: "apakah software masih jalan, saat ini, dengan perubahan terbaru dari semua orang?"

Empat prinsip, dan **alasan** tiap prinsip:

1. **Fail fast, fail loud.** Build rusak memblokir merge; stage termurah dan tercepat duluan; pipeline adalah *gate*, bukan saran. *Alasan:* kegagalan yang ditemukan cepat murah diperbaiki, dan kegagalan yang bisa diabaikan akan diabaikan.
2. **Immutable artifacts.** Build sekali per commit, lalu promosikan artifact yang sama (dikenali lewat content hash) dev → staging → prod. Perilaku per environment datang dari config dan secret yang di-inject saat deploy. *Alasan:* yang diuji harus persis yang dirilis; jika di-build ulang, hasilnya bisa berbeda.
3. **Fast feedback.** Hitungan menit; stage independen diparalelkan; dependency dan layer di-cache. *Alasan:* pipeline lambat akan di-bypass, sehingga tidak memberi keamanan.
4. **Trunk-based development dan small batches.** Short-lived branch, diff kecil, feature flag, hindari merge hell. *Alasan:* integrasi kecil dan sering lebih mudah dilacak penyebabnya. Feature flag memungkinkan pekerjaan yang belum selesai di-merge dengan aman karena disembunyikan di balik toggle.

Langkah 3. Apa saja yang dijalankan pipeline build-time

Urutannya: **Lint → Unit test → SAST → Build → SCA → Image scan → Publish.** Perhatikan pola: yang murah dan cepat di depan (prinsip fail fast); image scan harus setelah build karena memeriksa image yang sudah jadi. **Publish** hanya dijalankan dari branch main dan hanya bila semua stage sebelumnya lulus (contoh GitLab CI: `only: [main]`).

(a) Testing pyramid (dasar dari stage test):

Lapisan	Jumlah dan kecepatan	Fungsi
Unit	Paling banyak, detik; dependency di-mock atau di-stub	Memeriksa logika kecil secara terisolasi
Integration	Lebih sedikit, menit; dependency nyata	Menangkap SQL salah, serialization mismatch, wiring error di titik integrasi
Contract	Sedikit	Memastikan ekspektasi consumer cocok dengan API provider

Anti-pattern **ice cream cone**: berat di E2E atau manual test yang lambat dan brittle, sedikit unit test.

(b) SAST menganalisis source code **tanpa menjalankan aplikasi** (definisi: tambahan): injection flaws, hardcoded secrets, insecure deserialization, broken access logic. Contoh perbaikan: SQL injection dari string concatenation diperbaiki dengan **parameterized query** (nilai dikirim terpisah dari teks query).

(c) SCA menginventarisasi library pihak ketiga (direct dan transitive) dan mencocokkannya dengan database kerentanan (mis. NVD, National Vulnerability Database; kerentanan di dalamnya diberi ID standar bernama CVE; definisi NVD dan CVE: tambahan). *Alasan perlu:* 80 sampai 90 persen kode aplikasi modern adalah pihak ketiga.

(d) Image scan memeriksa seluruh layer (OS package, base image, dependency) setelah build dan sebelum push; blokir Critical dan High, track Medium dan Low. Base image minimal (distroless/slim) sudah mengecilkan attack surface.

Langkah 4. Menjaga artifact dari tampering: registry dan supply chain

- Artifact **content-addressed**: dirujuk lewat **digest** (hash isi), bukan tag `latest` yang bisa berpindah ke image lain. Karena itu image yang di-scan dijamin sama dengan yang di-deploy hanya bila dirujuk lewat digest.
- Registry mengatur siapa boleh push dan pull; credential build **tidak boleh** punya hak deploy ke production.
- **Provenance** mencatat source, pipeline, dan input artifact. **SLSA** membuktikan artifact dibangun oleh proses tepercaya dan tamper-resistant.

Tools: GitLab CI (`.gitlab-ci.yml`, job dalam satu stage paralel secara default), GitHub Actions (`.github/workflows`, event, marketplace, matrix), SonarQube (quality gate; new code dinilai terpisah dari legacy), Trivy (`--exit-code 1 --severity CRITICAL,HIGH` sebagai gate).

Pitfalls: alert fatigue (memblokir untuk setiap temuan Low), flaky tests (kadang lulus, kadang gagal, merusak kepercayaan), mutable `latest` tags, secrets in logs.

Langkah 5. Logika yang sama untuk data: kenapa data perlu diverifikasi

Biaya data buruk. Garbage in garbage out (diperbesar oleh ML), silent failures, compounding errors. **Reliability = Uptime × Data Quality**: pipeline bisa 100% available tetapi tetap tidak andal bila datanya cacat.

Dimensi data quality, dengan contoh pelanggaran pada event PayLend:

Dimensi DQ	Contoh di event PayLend
Validity	`currency_code = "USDD"` (bukan kode ISO 4217)
Completeness	amount negatif tanpa `transaction_type`
Uniqueness	`transaction_id` terkirim dua kali
Timeliness	timestamp sudah 3 hari atau ada di masa depan
Reasonability	amount 5.000.000
Integrity	`user_id` tidak punya baris di tabel users (satu versi slide menyebutnya Consistency)
Consistency	negara user berbeda antara dua service
Accuracy	amount berbeda 0,01 dari ledger

Slide menampilkan delapan dimensi, tetapi beberapa slide menyebut "6 core DQ dimensions" tanpa merinci enam yang mana.

Siklus perbaikan: Define rules → Measure → Analyze root cause → Improve → Control. **Perbaiki di sumber**: transformasi yang diam-diam "memperbaiki" nilai buruk justru menyembunyikan bug di producer.

Langkah 6. Di mana dan dengan apa data diverifikasi

Prinsip shift-left diterapkan ke data: validasi di **edge** (pintu masuk), bukan di model. Lapisan penegakannya, masing-masing menangkap jenis masalah yang berbeda:

1. **Avro enum** membuat nilai di luar daftar (mis. `"USDD"`) tidak bisa di-serialize, jadi tidak pernah sampai ke Kafka.
2. **Great Expectations** memvalidasi data sebelum mendarat (stream, file, DataFrame).
3. **dbt tests** memvalidasi tabel yang sudah ada di warehouse (`not_null`, `unique`, `accepted_values`, `relationships`; definisi dbt: tambahan).
4. **Compatibility check di CI**: Pull Request yang mengubah schema → CI job → registry compat check; bila incompatible, build gagal (registry menolak dengan HTTP 409 Conflict **bila compatibility mode aktif**).

Bila check gagal, apa yang dilakukan? Pilih strategi sesuai sifat kegagalan:

Strategi	Dipakai bila
Block	Defect tidak aman, mis. PII bocor
Quarantine	Kegagalan parsial dan independen; record buruk ke DLQ, yang baik diteruskan
Warn	Severity rendah atau rollout aturan baru
Circuit breaker	Failure rate melewati threshold, mis. producer rusak; hentikan semuanya

Bila 40% record masuk DLQ, itu bukan lagi masalah data quality, melainkan **incident**.

Langkah 7. Memverifikasi privacy: perlindungan PII

Setelah validitas, pertanyaannya: apakah data sensitif sudah aman? Teknik-tekniknya punya trade-off berbeda:

Teknik	Utilitas untuk ML	Compliance
Masking	Merusak utilitas	Bagus untuk tampilan
Tokenization (deterministik)	Menjaga referential integrity	Bagus untuk analitik
Encryption	Perlu dekripsi untuk ML	Terbaik untuk data at-rest
Deletion	Tidak ada data, tidak ada model	Terbaik untuk minimisasi

- **Presidio** mendeteksi dan mentransformasi PII (`AnalyzerEngine`, `AnonymizerEngine`), diterapkan **per field**.
- Token dibuat dengan **HMAC-SHA256 berkunci**: deterministik (input sama, token sama, jadi join tetap bisa) tetapi tidak bisa di-brute-force tanpa key. Key dari KMS atau secret store, tidak pernah di kode. **Hash tanpa salt** (mis. hash SSN) bisa di-brute-force dalam hitungan menit.
- **Urutan penting:** validasi **format** SSN **sebelum** tokenize, karena token selalu terlihat valid sehingga kesalahan format tidak akan terdeteksi lagi.
- Great Expectations dengan `result_format="BOOLEAN_ONLY"` agar hasil validasi tidak mencetak nilai buruk (mencegah **leaky evidence**).

Langkah 8. Memverifikasi model dan pipeline data

Model melewati **training and evaluation**, **validation** (memastikan layak dipakai), dan **reproducibility** sebelum di-deploy. Data pipeline di DataOps diuji lewat tahap Testing: data quality checks (integrity, accuracy, consistency) dan data validation (patuh standar dan constraint).

Jembatan ke tahap Release

Artifact sudah lolos semua gate: aman, teruji, dan dirujuk lewat digest. Pertanyaan berikutnya: **di mana artifact ini berjalan, siapa yang boleh mengubahnya, dan bagaimana dirilis tanpa membahayakan user?** Itu isi tahap Release.

5.2 Latihan: benarkan atau salahkan

Latihan V1: Prinsip CI

#	Pernyataan	Kunci	Alasan
1	Stage pipeline diurutkan dari yang paling murah dan cepat dulu (lint, lalu unit test, lalu integration test).	**Benar**	Fail fast.
2	Image di-build ulang untuk tiap environment agar sesuai environment itu.	**Salah**	Build once, promote everywhere; perbedaan environment lewat config dan secret yang di-inject saat deploy.
3	Pipeline merah boleh di-merge bila mendesak.	**Salah**	Pipeline adalah gate, bukan saran; tidak ada budaya “merge anyway”.
4	Feature flag memungkinkan pekerjaan yang belum selesai di-merge dengan aman daripada tinggal di long-lived branch.	**Benar**	Disembunyikan di balik toggle.

#	Pernyataan	Kunci	Alasan
5	Pipeline yang lambat cenderung di-bypass, sehingga tidak memberi keamanan.	**Benar**	Karena itu fast feedback: menit, paralel, cache.

Latihan V2: Shift-left dan testing pyramid

#	Pernyataan	Kunci	Alasan
1	Biaya memperbaiki bug: 1x di IDE atau pre-commit, 15x di CI, 100x+ di production.	**Benar**	Cost-of-defects curve.
2	Shift-left memindahkan pengecekan ke tahap akhir sebelum release.	**Salah**	Ke tahap awal (kiri) karena di sana perbaikan paling murah.
3	Unit test harus mayoritas, cepat, dan memakai mock atau stub untuk dependency.	**Benar**	Dasar piramida.
4	Integration test sebaiknya mengulang seluruh business logic.	**Salah**	Scope ketat di titik integrasi saja.
5	Ice cream cone (berat di E2E atau manual test, sedikit unit test) adalah bentuk testing yang sehat.	**Salah**	Itu anti-pattern; pipeline makan jam dan kegagalan sulit dilokalisasi.

Latihan V3: SAST, SCA, dan image scan

#	Pernyataan	Kunci	Alasan
1	SAST bisa menangkap hardcoded secrets dan SQL injection dari string concatenation.	**Benar**	Dua jenis temuan SAST.
2	SCA hanya memeriksa direct dependency.	**Salah**	Direct dan transitive.
3	Image scan dijalankan setelah build dan sebelum push ke registry.	**Benar**	Gate terakhir di build-time.
4	Deploy dengan tag `latest` menjamin yang di-deploy sama dengan yang di-scan.	**Salah**	Tag bisa berpindah; rujuk lewat digest.
5	SQL injection dari string concatenation diperbaiki dengan parameterized query.	**Benar**	Nilai dikirim terpisah dari teks query.

Latihan V4: Pipeline dan supply chain

#	Pernyataan	Kunci	Alasan
1	`trivy image --exit-code 1 --severity CRITICAL,HIGH` membuat job gagal bila ada temuan Critical atau High.	**Benar**	Berbasis exit code, jadi mudah dipakai sebagai gate.
2	Memblokir build untuk setiap temuan Low mencegah alert fatigue.	**Salah**	Justru memicu alert fatigue; blokir hanya Critical dan High.
3	Credential yang dipakai build boleh punya hak deploy ke production agar praktis.	**Salah**	Credential build tidak boleh punya hak deploy ke production.
4	SLSA membuktikan artifact dibangun oleh proses tepercaya melalui provenance.	**Benar**	Bukan sekadar scan kode sekali.

#	Pernyataan	Kunci	Alasan
5	Flaky test sebaiknya dibiarkan karena kadang lulus.	**Salah**	Merusak kepercayaan; perbaiki atau karantina segera.

Latihan V5: Pemetaan dimensi data quality

#	Pernyataan	Kunci	Alasan
1	`currency_code = "USDD"` melanggar Validity.	**Benar**	Bukan kode ISO 4217.
2	`transaction_id` yang terkirim dua kali melanggar Uniqueness.	**Benar**	Akibat delivery at-least-once.
3	`user_id` yang tidak ada di tabel users melanggar Accuracy.	**Salah**	Integrity (satu versi slide menyebut Consistency), bukan Accuracy.
4	Amount 5.000.000 untuk transaksi biasa dicek dengan Reasonability.	**Benar**	Range atau batas statistik.
5	Timestamp tiga hari lalu pada event real-time melanggar Completeness.	**Salah**	Timeliness.

Latihan V6: Shift-left testing data dan contract

#	Pernyataan	Kunci	Alasan
1	Data diuji di edge (pintu masuk), bukan baru di model.	**Benar**	Tangkap error di ingestion sebelum merusak feature store.
2	Transformasi yang diam-diam memperbaiki `USDD` adalah cara terbaik menangani masalah quality.	**Salah**	Perbaiki di sumber; perbaikan diam-diam menyembunyikan bug di producer.
3	Avro enum membuat `"USDD"` tidak bisa di-serialize sehingga tidak pernah sampai ke Kafka.	**Benar**	Lapisan domain contract.
4	Great Expectations untuk data sebelum mendarat, dbt tests untuk tabel yang sudah ada di warehouse.	**Benar**	Pemilihan tool.
5	Schema Registry tanpa compatibility mode di subject-nya sudah cukup menolak perubahan schema yang breaking.	**Salah**	Tanpa compatibility mode, perubahan breaking lolos dan contract tidak terlindungi; penolakan 409 hanya bila mode aktif.

Latihan V7: PII dan kegagalan check

#	Pernyataan	Kunci	Alasan
1	Tokenization deterministik menjaga referential integrity sehingga join tetap bisa dilakukan.	**Benar**	Input sama selalu menghasilkan token sama.
2	Masking menjaga utilitas data untuk ML.	**Salah**	Masking merusak utilitas.
3	Format SSN divalidasi sebelum tokenize karena token selalu terlihat valid.	**Benar**	Urutan penting.
4	Hash SSN tanpa salt aman dari brute force.	**Salah**	Bisa di-brute-force dalam hitungan menit; pakai HMAC berkunci.

#	Pernyataan	Kunci	Alasan
5	Quarantine dipakai bila kegagalan parsial dan independen; block bila defect-nya tidak aman.	**Benar**	Pemilihan strategi saat check gagal.

6. Release: Infrastruktur, Gate, dan Rilis

6.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: infrastruktur sebagai kode (IaC, Terraform) → tempat berjalan (Kubernetes) → siapa boleh melakukan apa dan apa yang boleh masuk (RBAC, admission, Kyverno) → bagaimana merilis ke user dengan risiko kecil (rolling, blue-green, canary) → bagaimana rilis dikendalikan lewat Git (GitOps) → lalu merilis model dan pipeline data.

Langkah 1. Infrastruktur sebagai kode (IaC)

Masalah. Server yang dibuat dengan klik-klik manual tidak bisa direproduksi, tidak bisa direview, dan lama-lama menyimpang dari yang diyakini tim (*configuration drift*: state aktual server tidak lagi sama dengan yang dicatat). **IaC**: infrastruktur didefinisikan di **file teks** dan dilacak di version control, bisa direview di pull request; satu source of truth yang reproducible. Ini perluasan Factor I ke lapisan infrastruktur.

Dua gaya menulisnya:

Aspek	Imperative (how)	Declarative (what)
Bentuk	Script langkah berurutan	Deskripsi end state yang diinginkan
Edge case	Harus ditangani eksplisit	Tool menghitung langkah aman
Dijalankan ulang	Sering gagal atau menduplikasi resource	Aman, tidak berubah bila kenyataan sudah sesuai

Konsep pendukung:

- **Idempotency**: menjalankan konfigurasi yang sama dua kali sama dengan sekali. Ini sifat logika reconciliation milik tool, bukan otomatis dimiliki setiap script.
- **Immutable infrastructure**: server diganti dengan yang baru dari definisi terbaru, bukan di-patch in place. Hasilnya: configuration drift hilang, dan rollback = deploy ulang definisi known-good.
- Pola provisioning lain: resource graph, modules, environments as data, state.

Langkah 2. Terraform: alur kerja yang aman

1. Tulis file `.tf`.
2. `terraform plan`: menghitung diff (create, change, destroy) **tanpa menyentuh apa pun**.
3. **Review** diff (di pull request).
4. `terraform apply`: mengeksekusi **persis** plan yang direview, jadi tidak ada kejutan antara plan dan apply.
5. **State** diperbarui.

Cara kerjanya: **provider** menerjemahkan blok HCL menjadi API call. Plan membandingkan tiga hal: **state file, infrastruktur aktual, dan desired config**. Karena state adalah ingatan Terraform, ia disimpan di **remote storage yang shared dan ter-lock** (agar dua orang tidak apply bersamaan). Kehilangan atau korupsi state adalah salah satu insiden Terraform paling umum.

Langkah 3. Kubernetes: tempat artifact berjalan

Masalah. Begitu jumlah container lebih dari segelintir, harus ada yang memutuskan penempatan dan pemulihannya; melakukannya manual tidak praktis. **Orchestrator** menangani scheduling, scaling, networking, dan **self-healing** (restart atau reschedule container yang crash). Kubernetes terus-menerus mencocokkan "apa yang berjalan" dengan "apa yang diminta"; pola deklaratif ini sama dengan Terraform.

Komponen (dari pintu masuk ke pelaksana):

- **Control plane**: **API server** (pintu depan, satu-satunya komponen yang dipercaya semua komponen lain), **scheduler** (memutuskan pod di node mana), **etcd** (source of truth state cluster), controller manager.
- **Worker node**: **kubelet** mengelola container secara lokal.
- Setiap perintah `kubectl` adalah HTTP request ke API server.

Objek inti: Pod (unit deploy terkecil), **Deployment** (mendeklarasikan N replica dan terus mencocokkan kenyataan), **Service** (alamat jaringan stabil di depan pod yang berubah-ubah).

Alat lokal: Kind (node sebagai container Docker, ringan, cocok untuk CI dan multi-node) vs **Minikube** (VM atau container single-node, dukungan driver dan addon lebih kaya).

Langkah 4. Gate keamanan cluster: siapa boleh apa, dan apa yang boleh masuk

Keamanan cluster dibahas mengikuti perjalanan sebuah request:

(a) Authentication lalu authorization (RBAC). Pertama *kamu siapa?* (authentication), lalu *boleh melakukan ini?* (authorization). **RBAC** mengatur authorization. **Role** = kumpulan permission di **satu namespace**; **RoleBinding** memberikannya ke user, group, atau service account; **ClusterRole** dan **ClusterRoleBinding** berlaku cluster-wide dan dipakai seperlunya karena terlalu luas.

(b) Admission controller. Mencegat request **setelah** authentication dan authorization, **sebelum** disimpan ke etcd. Dua jenis: **mutating** (bisa mengubah request, mis. inject sidecar, default resource limit) dan **validating** (hanya menerima atau menolak; di sinilah policy enforcement seperti Kyverno berada).

(c) Kyverno. Policy as code berbentuk YAML (`validate`, `mutate`, `generate`), berjalan sebagai admission controller. `validationFailureAction: Enforce` menolak pelanggar; `Audit` hanya mencatat, berguna untuk rollout policy baru tetapi anti-pattern bila dibiarkan selamanya (pelanggaran tidak pernah benar-benar diblokir).

(d) Non-root runtime. `runAsNonRoot: true` dan read-only root filesystem memperkecil blast radius container yang dibobol.

Merangkai semuanya: defense in depth. Beberapa lapis independen, sehingga bila satu lolos, yang berikutnya menangkap:

Lapis	Gate	Apa yang dicegah
1	**Build-Time Gate** (CI scans)	Kode dan image rentan
2	**Provisioning Gate** (Terraform plan dan review)	Infrastruktur berubah tanpa review
3	**Access Gate** (RBAC)	Aksi oleh pihak tidak berwenang
4	**Runtime Gate** (Kyverno admission)	Workload yang melanggar kebijakan

Pitfalls: manual cluster drift (`kubectl edit` di resource live), unlocked shared state, overly broad ClusterRoles (`cluster-admin`), audit-only forever.

Langkah 5. Merilis ke user dengan risiko kecil

Konsep dasar. Release engineering (GSRE Ch. 8): pemisahan tegas build, test, release; high velocity tanpa mengorbankan stabilitas; release pipeline sebagai produk. **Blast radius** = jumlah user atau sistem yang terdampak bila deployment gagal. Tujuannya: kecilkan blast radius dan MTTR (waktu pemulihan).

Deployment ≠ release. *Deployment* (teknis) = kode sampai ke production. *Release* (bisnis) = fitur diekspos ke user. **Feature flag** membuat kode di-deploy tetapi dorman, sehingga keduanya bisa dipisah.

Empat strategi, mengikuti urutan evolusi strategi deployment di slide:

Strategi	Pros	Cons
Big bang	Tidak ada keunggulan berarti; strategi legacy	Risiko dan blast radius tinggi
Rolling update (default Kubernetes)	Tanpa biaya infrastruktur tambahan, zero downtime	Dua versi berjalan bersamaan; buruk untuk perubahan schema DB yang tidak backward-compatible
Blue-green	Switch instan, rollback instan, green bisa dites dulu	Butuh 2x kapasitas infrastruktur
Canary	Blast radius minimal, user nyata	Butuh observability kuat dan traffic splitting

Progressive delivery = otomasi canary atau blue-green yang bergantung pada telemetry (metrics, logs, traces); **sistem** yang mengambil keputusan Go/No-Go, bukan manusia yang menatap dashboard. (Literatur juga membahas A/B testing, ring deployment, dan dark launching; tambahan.)

Rollback harus blameless dan frictionless; rollback yang butuh 30 menit berarti strategi deployment gagal. Tujuannya otomatis dan berbasis metrik (SLO error budget).

Langkah 6. GitOps: rilis dikendalikan lewat Git

Masalah. Bila orang bebas `kubectl apply` dari laptop, cluster menyimpang dari yang tercatat. **GitOps**: Git sebagai **single source of truth** untuk desired state; perubahan ke production wajib lewat commit dan Pull Request. **Empat prinsip**: declarative, versioned dan immutable, pulled automatically, continuously reconciled.

Push vs pull, dan kenapa pull unggul:

- **Push model**: CI server mem-build lalu mendorong ke cluster (`kubectl` atau `helm`), sehingga **CI butuh credential Kubernetes**.
- **Pull model**: **agent di dalam cluster** memantau Git dan menarik perubahan.

Keunggulan pull: **security** (CI tidak butuh credential K8s berhak tinggi), **network traversal** (cluster memulai koneksi outbound ke Git, aman di balik firewall atau NAT), **drift detection** (agent memantau cluster terus-menerus).

Alur pull, tiga fase:

1. **Build dan push**: CI/CD build, test, scan, push image ke registry, update manifest di Git (CI/CD tidak punya credential cluster).
2. **Pull dan reconcile**: agent mendeteksi perubahan, render dan bandingkan, apply, status Synced dan Healthy.
3. **Continuous drift detection**: deteksi actual ≠ desired, lalu auto-correct atau self-heal.

Tools:

- **Helm**: package manager Kubernetes; **chart** (kumpulan file resource), **release** (instance chart yang berjalan di cluster; beda dengan arti "release" di atas), repository; `values.yaml` untuk environment berbeda.
- **ArgoCD**: GitOps controller berbasis pull yang berjalan di dalam cluster (alternatif: Flux), dengan repo server dan application controller yang membandingkan desired dan actual state. **Sync policy**: Manual (memberi peringatan, menunggu persetujuan manusia) vs Automated Prune dan Self-Heal (langsung mengembalikan perubahan manual; bisa mengganggu developer yang debugging di production).

DevSecOps di release: image signing (Cosign atau Notary), **manifest scanning** (kube-score, Polaris), RBAC agar ArgoCD hanya deploy ke namespace tertentu, **jangan pernah commit raw secret** (pakai External Secrets Operator, Sealed Secrets, atau Vault), PR approvals.

Ilustrasi canary otomatis: Argo Rollouts menggeser 10% traffic ke canary; **Analysis Template** meng-query Prometheus ("error rate > 5%?"); bila gagal, rollout di-**abort otomatis** dan traffic kembali ke v1.

Best practice: immutable artifacts (promosikan **by digest**), Git sebagai absolute source of truth (hotfix lewat **emergency PR**; mencegah konfigurasi shadow IT), metric-driven automated rollback (SLI jelas). **Anti-pattern:** snowflake clusters, push-based "GitOps", big bang Friday deployments, canary without observability.

Langkah 7. Merilis model dan pipeline data

Model. Deployment dan serving: microservice dengan REST API untuk **online prediction**; embedded di edge atau mobile device; atau bagian dari batch prediction system. Seberapa matang proses rilisnya dinilai dengan **level MLOps** (Google Cloud):

Level	Ciri
0 Manual	Script-driven tanpa CI/CD, ML terputus dari operations, rilis jarang, tidak ada active performance monitoring
1 ML pipeline automation	Pipeline training otomatis dipicu trigger (continuous training), dengan feature store dan ML metadata store
2 CI/CD pipeline automation	Komponen pipeline itu sendiri di-build, di-test, di-package, dan di-deploy lewat CI/CD

Cara mengingat: level 0 = manual; level 1 = *training* diotomasi; level 2 = *pipeline-nya sendiri* ikut CI/CD.

Data. Task DataOps tahap Deployment: men-deploy pipeline ETL ke production, memasang monitoring dan logging untuk pipeline.

Jembatan ke tahap Operate

Perubahan sudah berjalan di production. Namun rilis bukan akhir: **bagaimana kita tahu ia sehat, dan apa yang dilakukan bila tidak?** Itu isi tahap Operate, yang hasilnya kembali menjadi masukan untuk Plan.

6.2 Latihan: benarkan atau salahkan

Latihan R1: IaC dan Terraform

#	Pernyataan	Kunci	Alasan
1	Declarative provisioning mendeskripsikan end state, dan tool yang menghitung langkahnya.	**Benar**	Aman dijalankan ulang.
2	Script `create_server.sh` yang dijalankan dua kali aman karena idempotent.	**Salah**	Shell script naif biasanya membuat server dua kali; idempotency milik logika reconciliation tool.
3	`terraform apply` mengeksekusi persis plan yang sudah direview.	**Benar**	Tidak ada kejutan antara plan dan apply.
4	State Terraform sebaiknya disimpan di laptop masing-masing anggota tim.	**Salah**	Remote storage yang shared dan ter-lock, agar apply tidak bertabrakan.
5	Immutable infrastructure berarti server diganti dengan yang baru dari definisi terbaru, bukan di-patch in place.	**Benar**	Menghilangkan configuration drift.

Latihan R2: Kubernetes dan RBAC

#	Pernyataan	Kunci	Alasan
1	`kubectl` berkomunikasi dengan cluster lewat HTTP request ke API server.	**Benar**	API server adalah pintu depan.
2	Scheduler adalah komponen yang menjalankan container di node.	**Salah**	Scheduler memutuskan penempatan pod; kubelet yang mengelola container di node.
3	Deployment mendeklarasikan N replica pod dan terus mencocokkan kenyataan dengan deklarasi.	**Benar**	Pola deklaratif yang sama dengan Terraform.
4	Role berlaku di seluruh cluster.	**Salah**	Role berlaku di satu namespace; ClusterRole yang cluster-wide.
5	RBAC mengatur authorization, bukan authentication.	**Benar**	Authentication (“kamu siapa”) terjadi lebih dulu.

Latihan R3: Admission control dan defense in depth

#	Pernyataan	Kunci	Alasan
1	Admission controller berjalan setelah authentication dan authorization, tetapi sebelum objek disimpan ke etcd.	**Benar**	Checkpoint terakhir.
2	Validating admission controller dapat meng-inject sidecar.	**Salah**	Itu tugas mutating; validating hanya menerima atau menolak.
3	Kyverno mode Audit hanya mencatat pelanggaran; membiarkannya Audit selamanya adalah anti-pattern.	**Benar**	Pelanggaran tidak pernah benar-benar diblokir.
4	Provisioning gate dalam defense in depth adalah RBAC.	**Salah**	Provisioning gate = Terraform plan dan review; RBAC adalah access gate.
5	Memberi `cluster-admin` sementara agar tidak terblokir adalah praktik yang baik.	**Salah**	Overly broad ClusterRoles merusak least privilege secara permanen.

Latihan R4: Strategi rilis

#	Pernyataan	Kunci	Alasan
1	Rolling update menjalankan dua versi bersamaan, sehingga buruk untuk perubahan schema DB yang tidak backward-compatible.	**Benar**	Limitation rolling update.
2	Blue-green memberi rollback instan, tetapi butuh 2x kapasitas infrastruktur.	**Benar**	Pros dan cons blue-green.
3	Canary tanpa observability tetap merupakan canary yang aman.	**Salah**	Canary butuh observability kuat; tanpa itu tidak ada dasar untuk memutuskan lanjut atau abort.
4	Big bang deployment memiliki blast radius rendah.	**Salah**	Blast radius tinggi (legacy).
5	Pada progressive delivery, manusia memutuskan Go/No-Go lewat dashboard.	**Salah**	Sistem yang memutuskan berdasarkan telemetry.

Latihan R5: Deployment, release, dan rollback

#	Pernyataan	Kunci	Alasan
1	Deployment memindahkan kode ke production (teknis), sedangkan release mengekspos fitur ke user (bisnis).	**Benar**	Keduanya dipisahkan.
2	Feature flag membuat kode yang sudah di-deploy tetap dorman.	**Benar**	Continuous deployment tanpa perubahan yang terus terlihat user.
3	Rollback yang butuh 30 menit menandakan strategi deployment sedang gagal.	**Benar**	Rollback harus frictionless.
4	Rollback sebaiknya mencari siapa yang bersalah agar tim lebih berhati-hati.	**Salah**	Harus blameless.
5	Rollback ideal otomatis dan berbasis metrik (SLO error budget).	**Benar**	Manusia buruk memantau dashboard selama deployment.

Latihan R6: GitOps

#	Pernyataan	Kunci	Alasan
1	Dalam GitOps, Git menjadi single source of truth untuk desired state.	**Benar**	Versioned, auditable, immutable.
2	Pada pull model, CI server butuh credential Kubernetes berhak tinggi.	**Salah**	Itu masalah push model; pada pull model agent di dalam cluster yang menarik perubahan.
3	Pull model bekerja di balik firewall atau NAT karena cluster memulai koneksi outbound ke Git.	**Benar**	Network traversal.
4	Hotfix darurat boleh langsung `kubectl apply` tanpa lewat Git.	**Salah**	Hotfix lewat emergency PR.
5	ArgoCD dengan sync Automated (prune dan self-heal) mengembalikan perubahan manual agar sesuai Git.	**Benar**	Bisa mengganggu developer yang sedang debugging di production.

Latihan R7: Helm, DevSecOps, dan anti-pattern

#	Pernyataan	Kunci	Alasan
1	Helm memakai `values.yaml` untuk mengelola environment berbeda.	**Benar**	Templating chart per environment.
2	Raw secret boleh di-commit ke Git selama repository private.	**Salah**	Jangan pernah; pakai External Secrets, Sealed Secrets, atau Vault.
3	"Release" di Helm adalah instance chart yang berjalan di cluster.	**Benar**	Beda dari "release" yang berarti mengekspos fitur ke user.
4	GitHub Actions yang menjalankan `kubectl apply` langsung ke cluster adalah GitOps yang benar.	**Salah**	Itu push-based "GitOps"; CI harus menyimpan credential cluster. Pakai pull model.
5	Image signing (Cosign atau Notary) memastikan hanya image yang sudah ditandatangani yang di-deploy.	**Benar**	Bagian DevSecOps di release pipeline.

Latihan R8: Level MLOps dan serving

#	Pernyataan	Kunci	Alasan
1	MLOps level 0 bersifat manual dan script-driven tanpa CI/CD, dengan ML terputus dari operations.	**Benar**	Rilis jarang dan tanpa monitoring aktif.
2	MLOps level 0 sudah punya active performance monitoring.	**Salah**	Level 0 lack of active performance monitoring.
3	Level 1 mengotomasi pipeline training yang dipicu trigger, dengan feature store dan ML metadata store.	**Benar**	Continuous training.
4	Level 1 sudah men-build, men-test, dan men-deploy komponen pipeline lewat CI/CD.	**Salah**	Itu ciri level 2.
5	Model bisa di-serve sebagai microservice REST untuk online prediction, di-embed di edge atau mobile, atau menjadi bagian batch prediction system.	**Benar**	Tiga opsi deployment dan serving.

7. Operate dan Observe: Menjalankan dan Mengawasi

7.1 Materi (baca berurutan)

Petunjuk

Alur bagian ini: kenapa perlu mengamati (tujuan monitoring) → bagaimana mengamati (metrik, alert, health check) → log sebagai bahan diagnosis → apa yang diamati pada pipeline data → apa yang diamati pada keamanan data → apa yang diamati pada model → hasilnya kembali ke Plan.

Langkah 1. Kenapa perlu mengamati: tujuan monitoring

Sistem yang berjalan terus berubah, dan kegagalan sering tidak terlihat sampai user mengeluh. **Monitoring** punya lima tujuan utama:

1. Mengidentifikasi **failure** dan fault penyebabnya, baik saat runtime maupun **postmortem** (analisis setelah failure terjadi).
2. Mengidentifikasi masalah performa.
3. Characterizing workload untuk **capacity planning** dan billing.
4. Mengukur reaksi user terhadap interface atau business offering.
5. Mendeteksi **intruder**.

Setiap tujuan butuh data dari sumber tertentu:

Goal monitoring	Sumber data
Failure detection	Application dan infrastructure
Performance degradation detection	Application dan infrastructure
Capacity planning	Application dan infrastructure
User reaction to business offerings	Application saja
Intruder detection	Application dan infrastructure

Cara membaca tabel: hanya reaksi user yang cukup dari data aplikasi, karena itu perilaku pengguna; sisanya perlu melihat aplikasi *dan* infrastruktur di bawahnya.

Tantangan monitoring: monitoring di tengah continuous changes; bottom-up vs top-down dan monitoring di cloud; arsitektur microservice (satu request melewati banyak service); volume besar data log terdistribusi.

Langkah 2. Bagaimana mengamati: metrik, alert, dan health check

- **Tools komprehensif** (Prometheus, Grafana, Datadog) untuk metrik real-time.
- **Key metrics**: CPU dan memory usage, network throughput, latency.
- **Alerts** saat threshold terlampaui, supaya masalah ditangani sebelum membesar.
- **Health checks**: Kubernetes mendukung **liveness probe** dan **readiness probe**; dengan itu hanya container sehat yang melayani traffic.
- **Otomasi monitoring**, bukan pengecekan manual.

Praktik performa operasional: monitor dan analisis performa, optimasi networking, orchestration tools (Kubernetes, Docker Swarm) untuk load balancing dan high availability, regular update dan patch.

Langkah 3. Log sebagai bahan diagnosis

Metrik memberi tahu *bahwa* ada masalah; log membantu menjelaskan *kenapa*. Praktik logging:

Praktik	Alasan
Centralize logs (ELK Stack atau Fluentd)	Log dari semua container dianalisis di satu tempat
Structured logging (JSON)	Mudah di-parse dan dianalisis
Level yang tepat (INFO, WARN, ERROR), hindari logging berlebihan	Logging berlebihan menurunkan performa dan menyulitkan pencarian
Amankan log (access control, signing, encryption)	Log bisa memuat informasi sensitif
Monitor volume log dengan log rotation dan retention policy	Mencegah masalah storage
Integrasikan dengan monitoring tools	Log dikorelasikan dengan performance metrics

Langkah 4. Mengamati pipeline data

Consumer lag = selisih antara offset terbaru di topic dan offset yang sudah diproses consumer; menunjukkan berapa pesan yang masih antre. Diamati dengan `kafka-consumer-groups --describe`. Pertanyaan operasional: berapa lag yang masih dapat diterima, dan siapa yang di-page. Terkait freshness SLI.

Pitfalls operasional pipeline: commit offset sebelum menulis (data loss), lupa key (urutan hilang), satu file kecil per event (small files), tanpa compatibility mode (contract tidak terlindungi), **retry tanpa batas untuk pesan buruk** (partition macet).

Dead Letter Queue (DLQ) menampung pesan yang gagal diproses (mis. schema violation) bersama metadata error di header; consumer lanjut ke pesan berikutnya sehingga partition tidak macet. Penanganan: alert dan triage manual, atau fix dan replay. DLQ **menyimpan record raw**, jadi sama sensitifnya dengan sumber: **enkripsi, batasi akses, beri masa kedaluwarsa**.

Pipeline lengkap sebagai gambaran akhir: Kafka → Presidio (sanitasi PII) → staging → Great Expectations (validasi) → feature store bila lolos, DLQ bila gagal → alert dan investigasi. Orkestrasi mencakup scheduling, retry, backfill.

Task DataOps: tahap Operations: data management, data governance, data security; tahap Feedback and Improvement: collect feedback, analyze performance, iterate and refine.

Langkah 5. Mengamati keamanan data

Dynamic masking: tabel sama, tampilan berbeda per role (mis. ML engineer dan analyst melihat `tkn_********`, fraud investigator melihat token penuh karena tujuannya terdokumentasi). **Audit**: catat siapa membaca kolom sensitif mana, kapan, dengan role apa, dan beri alert pada anomali.

Sudut pandang penyerang membantu menilai apakah perlindungan cukup:

Konsep	Arti
Quasi-identifiers	Kolom yang sendirian tidak mengenali orang, tetapi bila digabung bisa (mis. timestamp + amount + 4 digit kartu)
Linkage	Token yang stabil memungkinkan mengikuti histori satu orang lintas dataset
Weak tokens	Hash tanpa salt
Leaky evidence	Laporan validasi yang mencetak nilai raw

Data poisoning adalah masalah quality sekaligus security; validation gate juga berfungsi sebagai security control.

Langkah 6. Mengamati model

Model health monitoring dan governance mengawasi model yang di-serve. Karena **model drift** (performa turun saat data nyata berubah) bisa terjadi walaupun service tetap hidup, performa model harus dipantau secara teratur, bukan hanya uptime. Hasil monitoring dapat **memicu iterasi baru** (retraining), kembali ke data analysis (**model re-training and iteration**). **Continuous Training** adalah retraining otomatis yang dipicu data baru atau performance decay.

Langkah 7. Menutup lingkaran: kembali ke Plan

Lifecycle tidak berhenti di Operate. **Feedback** dari monitoring, incident, dan pengguna kembali ke Plan, sesuai loop DevOps (Operate → Observe → Continuous feedback) dan panah feedback di lifecycle XOps. Inilah alasan semua tahap sebelumnya diotomasi: supaya satu putaran dari ide sampai feedback bisa diulang cepat, aman, dan terukur, sesuai prinsip "feedback" Fowler di tahap Fondasi.

7.2 Latihan: benarkan atau salahkan

Latihan O1: Monitoring

#	Pernyataan	Kunci	Alasan
1	Reaksi user terhadap business offerings cukup dipantau dari data aplikasi.	**Benar**	Satu-satunya goal yang sumber datanya hanya aplikasi.
2	Intruder detection cukup memakai data infrastruktur.	**Salah**	Butuh data aplikasi dan infrastruktur.
3	Capacity planning memakai data aplikasi dan infrastruktur.	**Benar**	Juga untuk billing.
4	Postmortem adalah analisis yang dilakukan setelah failure terjadi.	**Benar**	Berbeda dengan analisis saat runtime.
5	Monitoring tidak menjadi tantangan pada sistem yang di-deploy terus-menerus, karena monitoring bersifat statis.	**Salah**	Monitoring under continuous changes adalah salah satu tantangan.

Latihan O2: Logging dan health check

#	Pernyataan	Kunci	Alasan
1	Log terpusat (ELK atau Fluentd) memudahkan pengelolaan dan analisis log dari semua container.	**Benar**	Centralize logs.
2	Structured logging (mis. JSON) lebih mudah di-parse dan dianalisis.	**Benar**	Format yang umum adalah JSON.
3	Log sebaiknya selalu dicetak di level paling detail agar tidak ada yang terlewat.	**Salah**	Pakai level yang tepat; logging berlebihan menurunkan performa dan menyulitkan pencarian.

#	Pernyataan	Kunci	Alasan
4	Kubernetes mendukung liveness probe dan readiness probe sebagai health check.	**Benar**	Memastikan container berfungsi benar.
5	Log rotation dan retention policy diterapkan untuk mencegah masalah storage.	**Benar**	Monitor volume log.

Latihan O3: Operasi data

#	Pernyataan	Kunci	Alasan
1	Consumer lag adalah selisih offset terbaru di topic dan offset yang sudah diproses consumer.	**Benar**	Menunjukkan berapa pesan yang masih antre.
2	Retry tanpa batas untuk pesan buruk aman karena akhirnya pasti berhasil.	**Salah**	Partition bisa macet; kirim ke DLQ.
3	DLQ menyimpan record buruk dalam bentuk raw, sehingga perlu dienkripsi dan dibatasi aksesnya.	**Benar**	Sama sensitifnya dengan sumber.
4	Bila 40% record masuk DLQ, itu masih masalah data quality biasa.	**Salah**	Itu incident.
5	Dynamic masking menampilkan kolom berbeda tergantung role pembaca, dan akses kolom sensitif diaudit.	**Benar**	Least privilege dengan audit.

Latihan O4: Reliability dan privacy saat operasi

#	Pernyataan	Kunci	Alasan
1	Pipeline dengan uptime 99.99% pasti andal.	**Salah**	Reliability = Uptime × Data Quality; bisa tetap membawa banyak data cacat.
2	Quasi-identifiers (mis. timestamp + amount + 4 digit kartu) yang digabung dengan data lain bisa me-re-identify seseorang.	**Benar**	Sudut pandang penyerang.
3	Laporan validasi yang mencetak nilai raw sensitif adalah leaky evidence.	**Benar**	Karena itu `BOOLEAN_ONLY`.
4	Token yang stabil membuat penyerang sulit melacak histori seseorang.	**Salah**	Token stabil justru memungkinkan linkage lintas dataset.
5	Data poisoning adalah masalah quality sekaligus security.	**Benar**	Validation gate juga security control.

Latihan O5: Operasi model

#	Pernyataan	Kunci	Alasan
1	Model health monitoring yang menunjukkan penurunan performa dapat memicu retraining dan iterasi baru.	**Benar**	Re-training and iteration.
2	Setelah model di-deploy, siklus ML selesai.	**Salah**	Monitoring dan retraining berlanjut.

#	Pernyataan	Kunci	Alasan
3	Continuous Training adalah retraining otomatis yang dipicu data baru atau performance decay.	**Benar**	Jawaban MLOps untuk CI.
4	Model drift tidak perlu dipantau bila uptime service baik.	**Salah**	Drift menurunkan performa meski service hidup; perlu regular monitoring.
5	Hasil dari production dipakai kembali sebagai feedback untuk memperbaiki model.	**Benar**	Feedback loop.

8. Latihan Campuran: Membenarkan atau Menyalahkan Praktik pada Kasus

Soal bentuk ini menggabungkan beberapa tahap dalam satu skenario. Nilai tiap praktik.

Kasus K1: Pipeline untuk service orders

Tim menyiapkan pipeline untuk service `orders` (Python, container) sampai berjalan di Kubernetes.

#	Pernyataan	Kunci	Alasan
1	Urutan lint, unit test, SAST, build, SCA, image scan, publish masuk akal karena stage murah dan cepat dijalankan lebih dulu.	**Benar**	Fail fast.
2	Untuk staging image di-build ulang dengan konfigurasi staging, untuk production di-build ulang lagi dengan konfigurasi production.	**Salah**	Build once; config dan secret di-inject saat deploy.
3	Publish hanya berjalan di branch main dan hanya bila semua stage sebelumnya lulus.	**Benar**	Gate, bukan saran.
4	CI menyimpan kubeconfig cluster production agar bisa `kubectl apply` setelah publish.	**Salah**	Push model; pakai pull model, CI hanya meng-update manifest di Git.
5	Manifest merujuk image lewat digest sehingga yang berjalan persis yang sudah di-scan.	**Benar**	Digest content-addressed.

Kasus K2: Event PayLend

Event berisi `ssn`, nomor kartu, `full_name`, `home_address`, amount `-50.00` tanpa `transaction_type`, dan `currency_code "USDD"`.

#	Pernyataan	Kunci	Alasan
1	Urutan: Presidio (sanitasi PII) → staging → Great Expectations (validasi) → feature store atau DLQ.	**Benar**	Pipeline end-to-end di slide.
2	Setelah `ssn` di-tokenize, event ini boleh lolos ke feature store karena masalah privacy sudah selesai.	**Salah**	Sanitasi hanya menangani privacy; `USDD` dan tipe kosong ditolak validasi dan di-quarantine.
3	Kartu di-mask jadi 4 digit terakhir dan `home_address` dihapus karena tidak dibutuhkan.	**Benar**	Masking dan minimisasi.
4	Record di DLQ boleh disimpan sebagai plaintext karena sudah gagal.	**Salah**	DLQ menyimpan data raw; enkripsi dan batasi akses.
5	Memperbaiki `USDD` secara diam-diam di tahap transformasi adalah solusi terbaik.	**Salah**	Perbaiki di sumber; perbaikan diam-diam menyembunyikan bug producer.

Kasus K3: Consumer ShopStream

Consumer landing membaca topic `shopstream.orders.v1` dan menulis file ke bronze di MinIO.

#	Pernyataan	Kunci	Alasan
1	Commit offset dilakukan setelah file tertulis (at-least-once), dan duplikat ditangani dengan nama object deterministik `topic-partition-startOffset`.	**Benar**	Effectively-once.
2	Karena at-least-once, consumer cukup di-restart dan duplikat tidak mungkin terjadi.	**Salah**	Crash antara menulis dan commit menghasilkan duplikat.
3	Event tiap order memakai key `order_id` agar `OrderPlaced` dan `PaymentAuthorized` order yang sama masuk partition yang sama.	**Benar**	Urutan terjaga.
4	Dengan 3 partition, menjalankan 6 consumer dalam satu group membuat semuanya bekerja paralel.	**Salah**	Scaling maksimal sebanyak jumlah partition.
5	Pesan rusak diarahkan ke `shopstream.orders.dlq` dengan metadata error, lalu consumer lanjut.	**Benar**	Mencegah partition macet.

Kasus K4: Aplikasi monolit di VM

Aplikasi menyimpan sesi user di memori, konfigurasi ditulis di kode, dan dijaga satu tim khusus. Akan dipindah ke container.

#	Pernyataan	Kunci	Alasan
1	Sesi dipindahkan ke backing service agar aplikasi stateless.	**Benar**	Factor VI; kriteria containerize.
2	Konfigurasi tetap di kode agar konsisten di semua environment.	**Salah**	Factor III: config di environment variables.
3	Image akhir dibuat lewat multi-stage build agar tools build tidak terbawa ke production.	**Benar**	Image lebih ramping.
4	Bug kecil diperbaiki dengan masuk ke container yang berjalan dan mengedit file.	**Salah**	Immutable: build image baru.
5	Container dijalankan sebagai non-root dengan read-only root filesystem.	**Benar**	Memperkecil blast radius.

Kasus K5: Model churn

Tim data science melatih model churn di notebook dan mengirim file model ke tim operations. Model jarang diperbarui dan tidak dipantau.

#	Pernyataan	Kunci	Alasan
1	Tim ini berada di MLOps level 0.	**Benar**	Manual, ML terputus dari operations, tanpa monitoring aktif.
2	Mengotomasi pipeline training yang dipicu trigger (data baru atau performa turun), dengan feature store dan ML metadata store, membawa tim ke level 1.	**Benar**	Continuous training.

#	Pernyataan	Kunci	Alasan
3	Level 2 dicapai dengan menambah feature store.	**Salah**	Feature store sudah ada di level 1; level 2 menambah CI/CD untuk komponen pipeline.
4	Pipeline DataOps yang menjamin data andal tidak relevan untuk retraining.	**Salah**	Data as a foundation.
5	Performance monitoring di production yang memicu trigger menutup loop feedback.	**Benar**	Model re-training and iteration.

9. Essay per Tahap

Rumus menjawab essay: (1) **tahap** mana yang sedang dibahas; (2) **masalah** di skenario; (3) **konsep atau prinsip** dengan nama istilahnya; (4) **rancangan** langkah demi langkah; (5) **alasan, trade-off, dan anti-pattern** yang dihindari.

Essay E1 (Fondasi): Automated dan Frequent

Soal. Sebuah tim merilis aplikasi tiga bulan sekali. Deploy dilakukan dengan menyalin file ke server dan mengubah konfigurasi production secara manual. Setiap rilis sering gagal dan rollback sulit. Analisis masalahnya dengan pandangan Fowler dan usulkan perbaikannya.

- **Masalah**: anti-pattern Fowler muncul sekaligus: deploy manual dan manual configuration management production. Rilis jarang membuat **delta** perubahan besar.
- **Kenapa gagal**: tanpa otomasi, proses tidak repeatable (hasil berbeda karena software, konfigurasi, environment, dan proses berubah). Delta besar berarti risiko besar, rollback sulit, feedback lambat.
- **Perbaikan**: automate almost everything lewat deployment pipeline (commit stage, acceptance, capacity, manual testing, release); keep everything in version control; rilis lebih sering dengan delta kecil (if it hurts, do it more frequently); build quality in; done means released; everybody responsible; continuous improvement.
- **Hasil**: feedback dari tiap perubahan datang cepat dan ditindaklanjuti; benefit: empowering teams, reducing errors, lowering stress, deployment flexibility. Ini juga inti DevOps: menghapus silo lewat loop discover sampai continuous feedback.

Essay E2 (Plan): Service yang melanggar 12-factor

Soal. Service `orders` menyimpan password database langsung di kode, membaca tabel milik service `inventory` secara langsung, dan menyimpan isi keranjang belanja di memori proses. Evaluasi desain ini memakai 12-Factor App dan pola microservices, lalu usulkan perbaikannya.

- **Password di kode** melanggar **Factor III**: secret bocor ke version control dan kode yang sama tidak bisa jalan di dua environment. Perbaikan: baca dari environment variable; `.env` lokal yang tidak masuk Git; ConfigMaps dan Secrets di Kubernetes.
- **Membaca tabel service lain** melanggar **database-per-service** dan merusak encapsulation. Perbaikan: `inventory` memiliki `inventory_db` sendiri; `orders` mengakses lewat API (client lewat **API Gateway**) atau event (tambahan); konsistensi antar-service dengan saga atau eventual consistency.
- **Keranjang di memori** melanggar **Factor VI**: proses harus stateless, data persisten di **backing service** (Factor IV). Tanpa itu, **Factor IX (Disposability)** gagal karena orchestrator tidak bisa kill dan reschedule container tanpa kehilangan data, dan **Factor VIII** (scale out lewat stateless process) menjadi sulit.
- **Batas service**: pecah per **business capability**, satu tim memiliki service end to end. Ingat **Conway's Law**: batas yang buruk menghasilkan distributed monolith; config separation saja tidak mengatasi blast radius.
- **Hasil**: image menjadi 12-factor artifact (tanpa baked-in config), dibangun sekali lalu dipromosikan; lingkungan lokal direproduksi dengan Docker Compose (service + db + MinIO) sesuai **Factor IV**, sekaligus mendukung dev/prod parity (Factor X; tambahan).

Essay E3 (Build): Migrasi aplikasi ke container

Soal. Sebuah aplikasi monolit berjalan di VM. Aplikasi menyimpan sesi user di memori, konfigurasi ditulis langsung di kode, dan dijaga satu tim khusus. Perusahaan ingin memindahkannya ke container. Nilai kesiapannya, jelaskan langkah migrasi, dan sebutkan best practice yang wajib diterapkan.

- **Penilaian kesiapan**: sesi di memori → **tidak stateless**, pindahkan ke backing service; konfigurasi di kode → melanggar Factor III, pindahkan ke environment variable; monolit → nilai dulu cohesionnya (kriteria: high cohesion), pertimbangkan pecah per business capability; dijaga tim khusus → kriteria menyebut app sebaiknya **tidak** dijaga specialized team.
- **Langkah migrasi**: discovery → requirement assessment (pecah atau tidak, risiko, prinsip migrasi) → container infrastructure design → containerize (Dockerfile per service) → testing → deployment dan rollout, diulang per bagian.
- **Best practice**: security (minimal base image, scan rutin, non-root, immutable image, TLS antar-container, network segmentation, signature, registry aman); performance (multi-stage build, resource requests dan limits, caching, health check, orchestration); logging dan monitoring (centralized structured logging, level tepat, rotation dan retention, alert pada CPU, memory, latency).
- **Hasil**: "works on my machine" hilang karena image identik dan reproducible di lokal, CI, production.

Essay E4 (Build, data): Pipeline event ShopStream

Soal. `orders-api` sebelumnya mengirim data lewat export CSV manual. Rancang pipeline streaming dari `orders-api` ke data lake. Jelaskan desain topic, partisi dan key, delivery semantics, cara menghindari duplikat, penanganan pesan rusak, dan pengelolaan perubahan schema.

- **Masalah awal**: CSV manual stale, bergantung pada orang, tanpa schema.
- **Desain**: `orders-api` mem-publish event Avro (`OrderPlaced`, `PaymentAuthorized`, dan lain-lain) ke topic `shopstream.orders.v1` dengan key `order_id` dan beberapa partition; dead-letter topic `shopstream.orders.dlq`.
- **Ordering**: key `order_id` membuat event satu order masuk partition yang sama; ordering hanya dalam satu partition.
- **Consumer group landing**: tiap partition dibaca satu consumer, scaling sampai jumlah partition. Tulis ke **bronze** MinIO dengan layout `dt=/hour=` dan buffer agar tidak small-file.
- **Delivery**: **at-least-once** (proses lalu commit setelah file tertulis). Duplikat dicegah dengan **idempotency**: nama object deterministik atau dedupe `event_id`; producer `enable.idempotence=true`, `acks=all` → **effectively-once**.
- **Pesan rusak**: validasi; yang gagal ke **DLQ** dengan metadata error, consumer lanjut; DLQ di-alert, di-triage, diperbaiki, lalu di-replay.
- **Schema evolution**: Schema Registry dengan compatibility mode; field baru optional `["null","string"]` default null; compatibility check di CI menolak perubahan breaking (409).
- **Verifikasi dan keamanan** (detail lab ShopStream, catatan 25): reconcile jumlah event diproduksi dengan jumlah event di lake; kredensial lewat env atau secret; image consumer di-scan Trivy; production butuh TLS dan topic ACL; akses bucket bronze dibatasi karena berisi PII.

Essay E5 (Verify): Merancang CI pipeline build-time

Soal. Rancang CI pipeline untuk service Python berbasis container. Sebutkan urutan stage, gate di tiap stage, dan alasan urutannya. Jelaskan bagaimana pipeline itu memenuhi prinsip CI.

Urutan	Stage	Gate	Tool
1	Lint	Style dan syntax lulus	flake8
2	Unit test	Semua lulus, coverage	pytest
3	SAST	Tidak ada temuan Critical atau High (quality gate)	SonarQube
4	Build	Image ter-build sekali per commit	docker build
5	SCA	Tidak ada CVE Critical atau High pada dependency	SCA, Trivy
6	Image scan	Tidak ada Critical atau High di image	`trivy image --exit-code 1`
7	Publish	Hanya di main dan bila semua lulus; push by digest	registry

- **Fail fast, fail loud**: stage termurah dan tercepat duluan; pipeline merah memblokir merge.
- **Immutable artifacts**: build sekali, promosi dev → staging → prod lewat digest; config dan secret di-inject saat deploy.
- **Fast feedback**: lint, unit test, SAST bisa paralel; cache dependency dan layer.
- **Trunk-based dan small batches**: branch pendek dan feature flag. **Testing pyramid**: banyak unit test, integration test seperlunya, contract test untuk API antar service.
- **Supply chain**: registry content-addressed, credential build tanpa hak deploy production, provenance menuju SLSA.
- **Hindari**: alert fatigue (blok hanya Critical dan High), flaky test, tag `latest`, secret di log.

Essay E6 (Verify, data): Mengamankan dan memvalidasi event PayLend

Soal. Diberikan event PayLend berisi `ssn`, `credit_card_num`, `full_name`, `home_address`, `transaction_amount = -50.00` tanpa `transaction_type`, dan `currency_code = "USDD"`. Identifikasi masalah quality dan privacy-nya, lalu rancang pipeline yang menanganinya secara otomatis.

- **Quality**: `USDD` melanggar **Validity** (bukan ISO 4217); amount negatif tanpa tipe melanggar **Completeness**; bila event terkirim dua kali, **Uniqueness**.
- **Privacy dan security**: `ssn`, `full_name`, `home_address` adalah PII; `credit_card_num` adalah PCI. Klasifikasikan lewat metadata (catalog, glossary, tags), lalu policy engine menerapkan kontrol.
- **Kontrol per field**: `ssn` → **tokenize** (HMAC berkunci dari secret store, menjaga join); kartu → **mask** jadi 4 digit terakhir; `home_address` → **delete**; `full_name`: diagram slide menulis "keep, low risk" tetapi payload hasil dan tabel before-after men-drop-nya, sebut keduanya dan pilih drop bila tidak dibutuhkan.
- **Shift-left dan contract**: Avro enum untuk `currency_code`; contract berisi domain, semantics (`amount` tidak negatif, `transaction_type` ada di charge atau refund), SLA freshness, owner.
- **Pipeline**: Kafka → **Presidio** (validasi format SSN dulu, lalu tokenize dan mask per field) → staging → **Great Expectations** → lolos ke feature store, gagal ke **DLQ** terenkripsi → alert.
- **Saat gagal**: quarantine untuk kegagalan parsial; block bila PII bocor; circuit breaker bila failure rate tinggi (40% ke DLQ = incident).

- **Prinsip**: perbaiki di sumber; privacy diperbaiki dengan menghapus, quality dengan menolak.

Essay E7 (Release): Provisioning dan pengamanan cluster

Soal. Tim ingin men-deploy image hasil CI ke Kubernetes. Jelaskan bagaimana Terraform, Kubernetes, RBAC, dan Kyverno dipakai, bagaimana desain ini membentuk defense in depth, dan kenapa pendekatan deklaratif aman dijalankan berulang.

- **Provisioning (Terraform)**: tulis `.tf` (namespace dan resource quota) → `terraform plan` (diff, direview di PR) → `terraform apply` (persis plan) → state di remote storage ter-lock. Deklaratif dan **idempotent**: dijalankan ulang tidak membuat duplikat karena tool merekonsiliasi desired dengan aktual.
- **Orkestrasi (Kubernetes)**: Deployment 3 replica dengan image by digest dan `securityContext.runAsNonRoot: true`; Service sebagai alamat stabil; self-healing me-reschedule pod yang crash.
- **Akses (RBAC)**: ServiceAccount `ci-deployer` dengan Role terbatas (get, list, create pods di namespace tertentu) lewat RoleBinding (contoh RBAC di catatan 23; bila memakai GitOps pull model, akun ini dipegang agent di dalam cluster, bukan server CI; tambahan); hindari ClusterRole lebar.
- **Runtime (Kyverno)**: ClusterPolicy `require-non-root` dengan `Enforce`, berjalan sebagai validating admission controller setelah authn dan authz, sebelum disimpan ke etcd. Rollout policy baru boleh mulai dengan `Audit`, lalu dinaikkan ke `Enforce`.
- **Defense in depth**: build-time gate (CI scan) → provisioning gate (plan dan review) → access gate (RBAC) → runtime gate (Kyverno). Tiap lapis independen.
- **Hindari**: `kubectl edit` di resource live (drift), state tanpa locking, `cluster-admin`, audit-only selamanya.

Essay E8 (Release): Strategi rilis fitur checkout

Soal. Sebuah e-commerce akan merilis fitur checkout baru yang berisiko. Bandingkan rolling update, blue-green, dan canary, pilih satu, dan jelaskan alur GitOps pull model serta rollback otomatisnya.

- **Perbandingan**: rolling menjalankan dua versi bersamaan dan berbahaya bila schema DB tidak backward-compatible; blue-green rollback instan tetapi 2x kapasitas; canary membatasi blast radius dengan user nyata tetapi butuh observability dan traffic splitting.
- **Pilihan: canary dengan progressive delivery**, karena fitur berisiko dan blast radius harus kecil. Blue-green sering dipilih untuk stateful app, jadi layak dipertimbangkan bila aplikasinya stateful.
- **Pisahkan deployment dari release**: kode di-deploy, fitur disembunyikan di balik **feature flag** sampai siap.
- **Alur GitOps pull**: CI build, test, scan, push image, lalu update manifest (`values.yaml`, tag atau digest) di Git lewat PR → ArgoCD di dalam cluster mendeteksi perubahan, render, compare, apply → **Argo Rollouts** membuat canary 10% → **Analysis Template** meng-query Prometheus (mis. error rate lebih dari 5%) → lolos: naikkan bertahap; gagal: **abort otomatis** ke versi stabil.
- **Kenapa pull**: CI tidak memegang credential cluster, aman di balik firewall, drift detection terus-menerus.
- **Keamanan**: image signing, manifest scanning, RBAC ArgoCD per namespace, secret lewat Sealed Secrets atau Vault, PR approval. **Hindari**: big bang Friday, canary tanpa observability, push-based "GitOps", snowflake cluster.

Essay E9 (Operate): Memantau dan mengembalikan service

Soal. Service `orders` yang berjalan di Kubernetes terasa lambat bagi pengguna setelah rilis. Rancang monitoring, logging, dan health check untuk mendeteksinya, serta mekanisme rollback.

- **Tujuan monitoring**: identifikasi failure (runtime dan postmortem), masalah performa, workload untuk capacity planning, reaksi user, intruder. Sumber data: **aplikasi dan infrastruktur** (reaksi user cukup dari aplikasi).
- **Metrik dan alert**: tools Prometheus, Grafana, atau Datadog; key metrics CPU dan memory, network throughput, latency; alert saat threshold terlampaui supaya ditangani sebelum membesar.
- **Health check**: liveness probe dan readiness probe di Kubernetes, sehingga hanya container sehat yang melayani traffic.
- **Logging**: log terpusat (ELK atau Fluentd), structured logging JSON, level yang tepat, log dia-mankan, log rotation dan retention, serta diintegrasikan dengan metrik agar log bisa dikorelasikan dengan performance metrics.
- **Tantangan**: sistem terus berubah, satu request melewati banyak service, volume log terdistribusi besar.
- **Rollback**: blameless dan frictionless; rollback 30 menit berarti strategi gagal. Idealnya otomatis dan berbasis metrik: canary dengan Analysis Template yang meng-query Prometheus (SLI jelas); bila melanggar, rollout di-abort dan traffic kembali ke versi stabil, sehingga blast radius hanya sebagian kecil user.
- **Feedback**: temuan masuk kembali ke Plan.

Essay E10 (Release dan Operate, model): Menaikkan level MLOps

Soal. Tim data science melatih model churn di notebook, lalu mengirim file model ke tim operations. Model jarang diperbarui dan performanya di production tidak dipantau. Tentukan level MLOps tim ini dan jelaskan langkah menuju level 1 dan 2, termasuk bagaimana monitoring menutup loop.

- **Level 0 (manual)**: script-driven tanpa CI/CD, ML terputus dari operations, rilis jarang, tidak ada monitoring performa aktif. Risiko: **model drift** tidak terdeteksi.
- **Ke level 1**: ubah eksperimen menjadi orchestrated experiment dan simpan sebagai source code; bangun **automated pipeline** di production (extraction, validation, preparation, training, eva-luation, validation) yang dipicu **trigger** (data baru atau penurunan performa; jadwal juga lazim, tambahan); tambah **feature store**, **ML metadata store**, dan **performance monitoring** (model registry sudah ada sejak level 0).
- **Ke level 2**: komponen pipeline itu sendiri lewat **CI** (build, test, package) dan **CD** (pipeline deployment); model serving lewat CD.
- **Menutup loop**: performance monitoring memicu trigger, pipeline training berjalan lagi (**continuous training**), model baru divalidasi lalu di-serve.
- **Peran DataOps**: data as a foundation; pipeline data yang andal (quality checks, validation) menjadi dasar retraining.

Essay E11 (Integratif): Satu perubahan menempuh seluruh alur

Soal. Tim menambah field `discount_code` ke event `OrderPlaced` dan merilis fitur kode diskon di service `orders`. Jelaskan perjalanan perubahan ini melalui tahap Plan, Build, Verify, Release, dan Operate, dengan praktik di tiap tahap.

- **Plan**: batas service per business capability; config lewat environment variable (Factor III); schema v2 dengan `discount_code` bertipe union `["null","string"]` default `null` (backward compatible); data contract diperbarui (semantics, owner); field diklasifikasi apakah PII.
- **Build**: feature branch short-lived dari main; Dockerfile dengan layer dependency di-cache dan multi-stage build; container non-root dan tanpa secret di kode; producer mem-publish event Avro ke Kafka dengan key `order_id`.
- **Verify**: CI menjalankan lint, unit test, SAST, build, SCA, image scan, publish by digest; PR yang mengubah schema melewati **compatibility check** (breaking ditolak dengan 409); validasi data di edge (Great Expectations, dbt tests); PII di-tokenize atau di-mask; check gagal → quarantine ke DLQ.
- **Release**: Terraform plan dan apply untuk namespace dan quota; Deployment dengan `runAsNonRoot`; RBAC dan Kyverno sebagai gate; manifest di-update di Git lewat PR, ArgoCD (pull) menyinkronkan; rilis **canary** 10% dengan Analysis Template; fitur disembunyikan di balik **feature flag** sehingga deployment terpisah dari release.
- **Operate**: monitoring (latency, error rate), health check, log terpusat, consumer lag dan DLQ dipantau, rollback otomatis bila metrik melanggar, audit akses kolom sensitif.
- **Feedback**: hasil monitoring dan insiden kembali ke Plan, sehingga siklus terus berulang.

10. Ringkasan Alur

Tahap	Tiga kata kunci (alat bantu hafalan, tambahan)	Yang harus langsung teringat
Fondasi	automated, frequent, feedback	Fowler: tiga anti-pattern, delapan prinsip; DevOps = practices, tools, culture; XOps = disiplin sama ke code, data, models
Plan	boundary, config, contract	12 faktor; microservice per business capability, database-per-service; schema + semantics + ownership; klasifikasi PII; security tanpa privacy mungkin, privacy tanpa security tidak
Build	reproducible, immutable, replayable	branch short-lived; container tanpa Guest OS; layer dependency sebelum kode; non-root; key menjaga urutan; at-least-once + idempotency; bronze tidak diubah
Verify	gate, shift-left, quarantine	1x, 15x, 100x+; build once by digest; pyramid bukan cone; SAST, SCA, image scan; dimensi DQ; tokenize setelah validasi format; block, quarantine, warn, circuit breaker
Release	declarative, progressive, pull	plan sebelum apply, state ter-lock; RBAC authorization, admission setelah authn dan authz; rolling, blue-green, canary; deployment $\neq$ release; GitOps pull; level MLOps 0, 1, 2
Operate	observe, rollback, feedback	goal monitoring dan sumber data; log terpusat dan terstruktur; consumer lag dan DLQ terenkripsi; Reliability = Uptime $\times$ Data Quality; model drift memicu retraining