REKAYASA PERANGKAT LUNAK TINGKAT LANJUT

Arsitektur Modern dan Skalabilitas Sistem

Ir. Soleman, S.Kom., M.Kom., MCE.
Dwi Retnoningsih, S.T., M.T.
Ir. Ade Davy Wiranata, S.Kom., M.Kom.
Hardika Khusnuliawati, S.Kom., M.Kom.

Rekayasa Perangkat Lunak Tingkat Lanjut : Arsitektur Modern dan Skalabilitas Sistem

UU No 28 tahun 2014 tentang Hak Cipta

Fungsi dan sifat hak cipta Pasal 4

Hak Cipta sebagaimana dimaksud dalam Pasal 3 huruf a merupakan hak eksklusif yang terdiri atas hak moral dan hak ekonomi.

Pembatasan Pelindungan Pasal 26

Ketentuan sebagaimana dimaksud dalam Pasal 23, Pasal 24, dan Pasal 25 tidak berlaku terhadap:

i. penggunaan kutipan singkat Ciptaan dan/atau produk Hak Terkait untuk pelaporan peristiwa aktual yang ditujukan hanya untuk keperluan penyediaan informasi aktual;
ii. Penggandaan Ciptaan dan/atau produk Hak Terkait hanya untuk kepentingan penelitian ilmu pengetahuan;
iii. Penggandaan Ciptaan dan/atau produk Hak Terkait hanya untuk keperluan pengajaran, kecuali pertunjukan dan Fonogram yang telah dilakukan Pengumuman sebagai bahan ajar; dan
iv. penggunaan untuk kepentingan pendidikan dan pengembangan ilmu pengetahuan yang memungkinkan suatu Ciptaan dan/atau produk Hak Terkait dapat digunakan tanpa izin Pelaku Pertunjukan, Produser Fonogram, atau Lembaga Penyiaran.

Sanksi Pelanggaran Pasal 113

1. Setiap Orang yang dengan tanpa hak melakukan pelanggaran hak ekonomi sebagaimana dimaksud dalam Pasal 9 ayat (1) huruf i untuk Penggunaan Secara Komersial dipidana dengan pidana penjara paling lama 1 (satu) tahun dan/atau pidana denda paling banyak Rp100.000.000 (seratus juta rupiah).
2. Setiap Orang yang dengan tanpa hak dan/atau tanpa izin Pencipta atau pemegang Hak Cipta melakukan pelanggaran hak ekonomi Pencipta sebagaimana dimaksud dalam Pasal 9 ayat (1) huruf c, huruf d, huruf f, dan/atau huruf h untuk Penggunaan Secara Komersial dipidana dengan pidana penjara paling lama 3 (tiga) tahun dan/atau pidana denda paling banyak Rp500.000.000,00 (lima ratus juta rupiah).

Rekayasa Perangkat Lunak Tingkat Lanjut : Arsitektur Modern dan Skalabilitas Sistem

Ir. Soleman, S.Kom., M.Kom., MCE
Dwi Retnoningsih, S.T., M.T.
Ir. Ade Davy Wiranata, S.Kom., M.Kom.
Hardika Khusnuliawati, S.Kom., M.Kom.

Rekayasa Perangkat Lunak Tingkat Lanjut : Arsitektur Modern dan Skalabilitas Sistem

ISBN : 978-634-7483-96-6

Penulis : Ir. Soleman, S.Kom., M.Kom., MCE, Dwi Retnoningsih, S.T., M.T., Ir. Ade Davy Wiranata, S.Kom., M.Kom., Hardika Khusnuliawati, S.Kom., M.Kom.

Url Buku : https://innovatixlabs.id/product/rekayasa-perangkat-lunak-3/

Desain Cover : Innovatix Labs Team

Ukuran : ix, 180, Uk: 15.5x23 cm

Cetakan Pertama : Februari 2026

Penerbit Takaza Innovatix Labs
Anggota Ikatan Penerbit Indonesia (IKAPI) No. 044/SBA/2023

Jl. Berlian Raya Blok M4, Pegambiran Ampalu Nan XX,
Lubuk Begalung, Kota Padang, Sumatera Barat
No Hp: +62 811 50321 47
Website: www.takaza.id
E-mail: bookspublishing@takaza.id

KATA PENGANTAR

Puji syukur ke hadirat Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga buku ini dapat disusun dan disajikan kepada pembaca. Buku ini disusun sebagai respons atas semakin meningkatnya kompleksitas pengembangan perangkat lunak modern, khususnya pada sistem berskala besar yang menuntut arsitektur yang matang, pengelolaan proyek yang terstruktur, serta integrasi teknologi mutakhir seperti komputasi awan, microservices, dan kecerdasan buatan. Materi dalam buku ini dirancang untuk memberikan pemahaman konseptual sekaligus praktis mengenai bagaimana sistem perangkat lunak dirancang, diimplementasikan, dan dikelola secara berkelanjutan dalam konteks organisasi dan industri modern.

Buku ini diharapkan dapat menjadi referensi akademik dan praktis bagi mahasiswa, dosen, peneliti, serta praktisi teknologi informasi yang ingin memperdalam pemahaman tentang arsitektur perangkat lunak, manajemen proyek skala enterprise, serta tantangan sosio-teknis yang menyertainya. Penulis menyadari bahwa buku ini masih memiliki keterbatasan, sehingga kritik dan saran yang konstruktif sangat diharapkan untuk penyempurnaan di masa mendatang. Semoga buku ini memberikan manfaat dan kontribusi nyata dalam pengembangan ilmu pengetahuan dan praktik rekayasa perangkat lunak.

Penulis

DAFTAR ISI

DAFTAR GAMBAR

BAB I
LANDASAN REKAYASA PERANGKAT LUNAK TINGKAT LANJUT

Perkembangan teknologi digital yang sangat cepat, termasuk kemajuan kecerdasan buatan (AI), telah mengubah cara perangkat lunak dirancang dan digunakan. Perangkat lunak saat ini tidak lagi sekadar menjalankan perintah, tetapi juga mampu menyesuaikan diri, menganalisis data, dan membantu pengambilan keputusan. Di saat yang sama, sistem harus melayani pengguna dalam jumlah besar, bekerja tanpa henti, dan tetap cepat serta andal. Kondisi ini membuat arsitektur perangkat lunak menjadi bagian yang sangat penting, karena dari sanalah ditentukan bagaimana sistem dapat berkembang, bertahan, dan beradaptasi terhadap perubahan.

Bab ini memberikan gambaran awal tentang rekayasa perangkat lunak tingkat lanjut sebagai dasar untuk memahami pembahasan pada bab-bab berikutnya. Pembaca diajak mengenal bagaimana rekayasa perangkat lunak berkembang seiring meningkatnya kompleksitas sistem, serta bagaimana peran arsitektur membantu menjaga kinerja, keamanan, dan skalabilitas. Selain aspek teknis, perangkat lunak juga dipahami sebagai bagian dari lingkungan sosial dan organisasi, di mana manusia, proses kerja, dan teknologi saling terkait. Dengan pemahaman ini, pembaca diharapkan dapat melihat hubungan yang jelas antara arsitektur modern, AI, dan kebutuhan sistem perangkat lunak masa kini.

A. Evolusi Rekayasa Perangkat Lunak Modern

Evolusi rekayasa perangkat lunak dapat dipahami secara kronologis sebagai respons terhadap perubahan kebutuhan teknologi, organisasi, dan masyarakat.

- **1950–1970-an (Era Awal Komputasi)**. Perangkat lunak dikembangkan untuk kebutuhan ilmiah dan militer pada sistem mainframe terpusat. Pengembangan masih bersifat pemrograman ad hoc tanpa metodologi baku, pengujian sistematis, atau fokus pemeliharaan, sehingga banyak proyek gagal akibat kompleksitas kode (*software crisis*).

- **1970–1990-an (Pendekatan Terstruktur)**. Meningkatnya penggunaan komputer di sektor bisnis dan pemerintahan melahirkan metodologi formal seperti *waterfall*. Fokus utama adalah dokumentasi, keandalan, dan prediktabilitas sistem, meskipun pendekatan ini kurang fleksibel terhadap perubahan kebutuhan.

- **1995–2005 (Era Internet dan Aplikasi Web)**. Internet mendorong lahirnya aplikasi web, e-commerce, dan sistem terhubung global. Tantangan bergeser ke kinerja, keamanan, dan ketersediaan layanan, sehingga arsitektur sistem mulai menjadi perhatian utama dalam rekayasa perangkat lunak.

- **2010-an (Service-Oriented dan Microservices)**. Pertumbuhan platform digital memicu peralihan dari sistem monolitik ke SOA dan microservices. Sistem dibangun sebagai kumpulan layanan kecil yang otonom untuk mendukung

skalabilitas, inovasi cepat, dan ketahanan terhadap kegagalan parsial.

- **2010-an–sekarang (Cloud-Native dan CI/CD)**. Komputasi awan memungkinkan infrastruktur elastis dan mendorong praktik CI/CD. Perangkat lunak dapat dirilis secara berkelanjutan dengan risiko lebih kecil, menyesuaikan tuntutan pasar yang dinamis.

- **2020-an–sekarang (AI-Driven Software Engineering)**. Kecerdasan buatan digunakan dalam pengujian otomatis, observability, prediksi kegagalan, dan *self-healing systems*. Evolusi ini didorong oleh kompleksitas sistem, skala data besar, dan kebutuhan akan sistem adaptif.

Rekayasa perangkat lunak berevolusi dari aktivitas pemrograman sederhana menjadi disiplin arsitektural yang berfokus pada skala, ketahanan, otomatisasi, dan kecerdasan sistem. Perangkat lunak kini dipahami sebagai sistem hidup yang terus beradaptasi dalam lingkungan teknologi dan bisnis yang dinamis.

B. Kompleksitas Sistem Skala Besar

Kompleksitas sistem skala besar dapat dipahami melalui platform e-commerce seperti Shopee. Bagi pengguna, sistem tampak sederhana mencari produk, melakukan pembayaran, dan menunggu pengiriman. Namun di balik antarmuka tersebut terdapat sistem perangkat lunak yang sangat kompleks, terdiri dari banyak komponen yang harus melayani jutaan pengguna secara bersamaan, memproses transaksi dalam jumlah besar, dan beroperasi tanpa henti. Kompleksitas ini

muncul karena sistem harus menjaga kinerja, konsistensi, dan ketersediaan layanan dalam skala masif.

Salah satu sumber utama kompleksitas adalah pemecahan sistem ke dalam banyak layanan yang masing-masing memiliki fungsi, aturan bisnis, dan basis data sendiri. Proses pembelian, misalnya, melibatkan koordinasi antara layanan pencarian, pembayaran, inventori, logistik, dan notifikasi. Ketika terjadi lonjakan trafik seperti *flash sale*, sistem harus mampu menyesuaikan kapasitas secara otomatis tanpa memicu kegagalan berantai. Tantangan tidak hanya terletak pada peningkatan kapasitas, tetapi juga pada menjaga kestabilan interaksi antar layanan yang saling bergantung.

Selain aspek teknis, kompleksitas sistem skala besar juga bersifat bisnis dan organisasional. Platform global harus menyesuaikan diri dengan regulasi, metode pembayaran, dan perilaku pengguna yang berbeda di setiap wilayah. Di sisi lain, pengembangan dan operasional sistem melibatkan banyak tim dengan tanggung jawab yang terdistribusi. Tanpa arsitektur yang jelas dan mekanisme koordinasi yang baik, kompleksitas teknis akan semakin diperparah. Oleh karena itu, rekayasa perangkat lunak modern menempatkan pengelolaan kompleksitas sebagai fokus utama melalui modularisasi, otomatisasi, dan praktik operasional yang matang.

C. Hubungan Arsitektur, Kinerja, dan Skalabilitas

Dalam sistem perangkat lunak modern, arsitektur, kinerja, dan skalabilitas merupakan tiga aspek yang saling bergantung. Arsitektur menentukan bagaimana komponen sistem disusun dan berinteraksi, kinerja mencerminkan kecepatan serta responsivitas sistem, sementara skalabilitas menunjukkan kemampuan sistem mempertahankan

kualitas layanan saat beban meningkat. Keputusan arsitektural yang diambil sejak awal pengembangan akan membentuk batas kemampuan sistem dalam menjaga kinerja dan beradaptasi terhadap pertumbuhan pengguna di masa depan.

Hubungan ini terlihat jelas pada platform e-commerce berskala besar. Ketika terjadi lonjakan pengguna pada periode promosi, sistem harus tetap mampu memproses permintaan secara cepat dan stabil. Pada arsitektur monolitik, peningkatan beban pada satu fungsi dapat memengaruhi seluruh sistem karena semua komponen saling terikat dalam satu kesatuan. Sebaliknya, arsitektur modular atau berbasis layanan memungkinkan peningkatan kapasitas dilakukan secara selektif pada komponen yang kritis, sehingga kinerja sistem secara keseluruhan tetap terjaga meskipun beban meningkat secara signifikan.

Selain struktur sistem, kinerja juga dipengaruhi oleh bagaimana arsitektur mengatur alur komunikasi dan pengelolaan data. Arsitektur yang tidak efisien dapat menimbulkan latensi tinggi dan kegagalan layanan, yang langsung dirasakan pengguna. Skalabilitas pun bukan sekadar menambah sumber daya, melainkan kesiapan arsitektur untuk tumbuh secara berkelanjutan. Oleh karena itu, arsitektur berfungsi sebagai fondasi strategis yang menentukan apakah sistem mampu berkembang, mempertahankan kinerja, dan mendukung inovasi dalam jangka panjang.

D. Perangkat Lunak sebagai Sistem Sosio-Teknis

Perangkat lunak modern tidak dapat dipahami hanya sebagai artefak teknis berupa kode dan infrastruktur. Dalam praktiknya, perangkat lunak selalu beroperasi dalam konteks manusia, organisasi,

dan lingkungan sosial tertentu. Karena itu, perangkat lunak lebih tepat dipandang sebagai sistem sosio-teknis, yaitu sistem yang terbentuk dari interaksi antara komponen teknis perangkat lunak, perangkat keras, dan data dengan komponen sosial seperti manusia, proses kerja, budaya organisasi, serta aturan dan kebijakan. Keberhasilan atau kegagalan sistem sering kali ditentukan bukan hanya oleh kualitas teknologinya, tetapi oleh cara sistem digunakan, dipahami, dan dikelola.

Pendekatan sosio-teknis sangat relevan pada sistem skala besar yang melibatkan banyak aktor dan kepentingan. Platform digital seperti e-commerce tidak hanya melayani pengguna akhir, tetapi juga penjual, mitra logistik, tim pengembang, tim operasi, dan regulator yang memiliki tujuan serta tingkat literasi teknologi berbeda. Perangkat lunak harus mampu menjembatani perbedaan ini agar ekosistem tetap berfungsi.

Keputusan arsitektur juga memiliki dampak sosial dan organisasional. Arsitektur modular dapat meningkatkan kemandirian tim, tetapi menuntut koordinasi, dokumentasi, dan budaya kolaborasi yang kuat. Tanpa kesiapan organisasi, solusi teknis yang canggih justru dapat menambah kompleksitas dan fragmentasi kerja.

Selain itu, adaptasi sistem terhadap perubahan regulasi dan perilaku pengguna bukan hanya soal perubahan kode, tetapi juga penyesuaian proses, pelatihan, dan komunikasi. Meski otomatisasi dan praktik seperti DevOps berkembang, peran manusia tetap krusial dalam keputusan strategis, penanganan insiden, dan pertimbangan etis. Dengan perspektif sosio-teknis, rekayasa perangkat lunak dapat menghasilkan sistem yang andal, skalabel, dan dapat diterima dalam

konteks sosial nyata. (Sommerville, 2016; Pressman & Maxim, 2020; Bass, Clements, & Kazman, 2022).

E. Tantangan Global dalam Rekayasa Perangkat Lunak

Rekayasa perangkat lunak modern berkembang dalam konteks global yang sangat dinamis. Sistem tidak lagi dibangun untuk lingkungan lokal, tetapi harus beroperasi lintas negara, budaya, regulasi, dan infrastruktur teknologi. Kondisi ini melahirkan berbagai tantangan global yang memengaruhi cara perangkat lunak dirancang, dikembangkan, dan dikelola. Tantangan-tantangan tersebut tidak berdiri sendiri, melainkan saling terkait dan semakin kompleks seiring dengan meningkatnya ketergantungan dunia terhadap sistem digital.

1. Skala dan Pertumbuhan Pengguna yang Masif

Perangkat lunak global harus mampu melayani jutaan hingga miliaran pengguna dengan pola penggunaan yang sangat beragam. Lonjakan trafik yang tidak terduga, seperti pada kampanye digital atau peristiwa global, menuntut sistem yang sangat skalabel dan tangguh. Kegagalan mengelola skala ini dapat menyebabkan gangguan layanan yang berdampak luas terhadap kepercayaan publik dan stabilitas bisnis.

2. Kompleksitas Sistem Terdistribusi

Sistem global umumnya dibangun sebagai sistem terdistribusi yang tersebar di berbagai wilayah geografis. Tantangan utama muncul dalam menjaga konsistensi data, mengelola latensi jaringan, serta memastikan keandalan layanan ketika sebagian sistem mengalami gangguan. Kompleksitas ini menuntut desain arsitektur yang matang dan pemantauan sistem yang berkelanjutan.

3. Keamanan dan Ancaman Siber Global

Ancaman keamanan bersifat lintas batas dan terus berkembang. Serangan siber tidak hanya menargetkan kelemahan teknis, tetapi juga manusia dan proses organisasi. Sistem perangkat lunak harus dirancang dengan pendekatan keamanan menyeluruh yang mencakup perlindungan data, identitas pengguna, serta ketahanan terhadap serangan skala besar.

4. Keragaman Regulasi dan Kepatuhan Hukum

Perangkat lunak global harus mematuhi berbagai regulasi yang berbeda di setiap negara, seperti perlindungan data pribadi, pajak digital, dan standar industri. Perbedaan regulasi ini menuntut fleksibilitas arsitektur dan proses pengembangan agar sistem dapat beradaptasi tanpa mengganggu layanan utama.

5. Kolaborasi Tim Global dan Distribusi SDM

Pengembangan perangkat lunak kini melibatkan tim yang tersebar di berbagai negara dan zona waktu. Tantangan muncul dalam koordinasi, komunikasi, dan penyelarasan standar kerja. Tanpa mekanisme kolaborasi yang efektif, kompleksitas teknis akan semakin diperparah oleh kompleksitas organisasi.

6. Kecepatan Inovasi dan Tekanan Pasar

Persaingan global menuntut inovasi yang cepat dan berkelanjutan. Perangkat lunak harus terus diperbarui untuk mengikuti perubahan kebutuhan pengguna dan perkembangan teknologi. Tekanan ini sering kali bertabrakan dengan kebutuhan akan stabilitas dan keandalan sistem, sehingga menimbulkan dilema dalam pengambilan keputusan teknis.

7. Etika, Privasi, dan Dampak Sosial Teknologi

Penggunaan data dalam skala besar serta integrasi kecerdasan buatan menimbulkan isu etika dan privasi yang serius. Rekayasa perangkat lunak tidak hanya bertanggung jawab secara teknis, tetapi juga secara sosial. Keputusan desain sistem dapat berdampak langsung pada hak pengguna, keadilan, dan kepercayaan masyarakat.

8. Keberlanjutan dan Efisiensi Energi

Sistem digital global mengonsumsi sumber daya energi yang besar. Tantangan keberlanjutan mendorong rekayasa perangkat lunak untuk memperhatikan efisiensi energi, optimasi infrastruktur, dan dampak lingkungan dari sistem yang dibangun. Aspek ini semakin penting dalam konteks agenda global menuju pembangunan berkelanjutan.

F. Arah Riset dan Inovasi Rekayasa Perangkat Lunak Tingkat Lanjut

Perkembangan sistem digital yang semakin kompleks mendorong rekayasa perangkat lunak tingkat lanjut untuk memperluas fokus risetnya. Riset tidak lagi terbatas pada teknik pemrograman, tetapi diarahkan pada pembangunan sistem yang adaptif, tangguh, aman, dan berkelanjutan dalam skala besar dan dinamis.

Salah satu arah utama riset adalah arsitektur adaptif dan self-adaptive systems, yaitu sistem yang mampu menyesuaikan konfigurasi dan perilaku secara otomatis terhadap perubahan lingkungan dan beban kerja tanpa intervensi manusia (Weyns et al., 2012). Sejalan dengan itu, integrasi kecerdasan buatan semakin menonjol dalam siklus hidup perangkat lunak, mulai dari desain, pengujian, hingga

operasi sistem, sehingga perangkat lunak menjadi lebih cerdas dan proaktif (Amershi et al., 2019; Zhang et al., 2022).

Riset juga berkembang pada software engineering for systems of systems, di mana sistem modern saling terhubung namun berevolusi secara mandiri. Fokusnya adalah menjaga interoperabilitas dan kendali arsitektural pada domain seperti smart city dan infrastruktur digital (Sommerville et al., 2012). Selain itu, isu reliability dan resilience mendorong inovasi pada site reliability engineering dan chaos engineering untuk memastikan sistem tetap beroperasi meskipun terjadi kegagalan (Basiri et al., 2016; Beyer et al., 2016).

Kesadaran lingkungan memperkuat riset green software engineering, yang menekankan efisiensi energi dan keberlanjutan sistem digital (Hilty & Aebischer, 2015; Lago et al., 2021). Di sisi lain, pendekatan human-centered dan socio-technical menegaskan peran manusia, organisasi, dan etika dalam keberhasilan sistem perangkat lunak modern (Baxter & Sommerville, 2011; Storey et al., 2020).

BAB II
PRINSIP DASAR ARSITEKTUR PERANGKAT LUNAK MODERN

Dalam sistem perangkat lunak modern, arsitektur memegang peran sentral yang menentukan bagaimana sebuah sistem dibangun, dikembangkan, dan dipertahankan dalam jangka panjang. Arsitektur bukan sekadar gambaran struktur teknis, tetapi kerangka berpikir yang menghubungkan kebutuhan bisnis, batasan teknologi, dan kualitas sistem yang diharapkan. Keputusan arsitektural yang diambil pada tahap awal akan memengaruhi kinerja, skalabilitas, keandalan, serta kemudahan perubahan sistem di masa depan. Oleh karena itu, pemahaman terhadap prinsip dasar arsitektur perangkat lunak menjadi fondasi penting sebelum membahas bentuk-bentuk arsitektur modern yang lebih kompleks.

Bab ini membahas prinsip-prinsip dasar arsitektur perangkat lunak modern yang digunakan untuk mengelola kompleksitas sistem skala besar. Pembahasan mencakup konsep dan peran arsitektur, berbagai gaya arsitektur dan pola desain, atribut kualitas sistem, serta trade-off yang muncul dalam proses perancangan. Selain itu, dibahas pula pentingnya modularitas, arsitektur berlapis, dan evaluasi arsitektur sebagai upaya memastikan sistem tetap relevan, andal, dan mudah dikembangkan. Dengan memahami prinsip-prinsip ini, pembaca diharapkan memiliki kerangka berpikir arsitektural yang kuat sebagai bekal untuk memahami arsitektur berbasis layanan, cloud-native, dan sistem skalabel pada bab-bab selanjutnya.

A. Konsep dan Peran Arsitektur Perangkat Lunak

Arsitektur perangkat lunak dapat dipahami dengan mudah melalui analogi arsitektur bangunan. Sebelum sebuah gedung dibangun, seorang arsitek terlebih dahulu merancang struktur dasar: fondasi, jumlah lantai, pembagian ruang, jalur listrik, dan sistem keamanan. Rancangan ini tidak hanya menentukan bentuk gedung, tetapi juga menentukan apakah gedung tersebut kuat, aman, nyaman digunakan, dan mudah direnovasi di masa depan. Dalam konteks perangkat lunak, arsitektur memiliki peran yang sama, yaitu merancang struktur dasar sistem sebelum kode ditulis secara rinci.

Seperti halnya bangunan, perangkat lunak terdiri dari banyak "ruang" dengan fungsi berbeda. Dalam sistem perangkat lunak, ruang-ruang tersebut dapat dianalogikan sebagai modul, layanan, atau komponen. Arsitektur menentukan komponen apa saja yang ada, bagaimana mereka saling terhubung, dan aturan interaksi di antara mereka. Jika sebuah bangunan dirancang tanpa perencanaan yang baik, misalnya dengan fondasi yang lemah atau jalur utilitas yang tidak jelas, maka bangunan tersebut akan sulit diperluas dan rawan kerusakan. Demikian pula, perangkat lunak tanpa arsitektur yang jelas akan sulit dikembangkan, mudah mengalami kesalahan, dan mahal untuk diperbaiki.

Peran arsitektur juga terlihat jelas ketika terjadi perubahan kebutuhan. Gedung yang dirancang dengan struktur modular akan lebih mudah direnovasi atau ditambah lantai tanpa harus merombak seluruh bangunan. Dalam perangkat lunak, arsitektur yang baik memungkinkan penambahan fitur, peningkatan kapasitas, atau penggantian teknologi tertentu tanpa mengganggu keseluruhan sistem. Arsitektur inilah yang menentukan seberapa fleksibel sistem

menghadapi perubahan, baik dari sisi teknologi maupun kebutuhan pengguna.

Selain itu, arsitektur bangunan harus mempertimbangkan faktor non-fungsional seperti keamanan, kenyamanan, dan efisiensi energi. Gedung yang indah tetapi tidak aman atau boros energi akan gagal memenuhi tujuannya. Hal yang sama berlaku dalam perangkat lunak. Arsitektur perangkat lunak menentukan kinerja, skalabilitas, keandalan, dan keamanan sistem. Aspek-aspek ini tidak dapat "ditambal" di akhir pembangunan, tetapi harus dirancang sejak awal melalui keputusan arsitektural yang tepat.

Dengan analogi ini, arsitektur perangkat lunak dapat dipahami sebagai cetak biru (*blueprint*) sistem yang menjadi acuan bagi seluruh proses pengembangan. Kode adalah bahan bangunan, pengembang adalah pekerja konstruksi, dan arsitektur adalah rancangan yang memastikan semua bagian sistem bekerja sebagai satu kesatuan yang kokoh. Tanpa arsitektur yang baik, baik bangunan maupun perangkat lunak berisiko runtuh ketika menghadapi beban, perubahan, dan tuntutan jangka panjang.

WhatsApp: Mengapa arsitekturnya baik?

WhatsApp melayani miliaran pesan harian dengan tim yang relatif kecil.

Analogi bangunan:

Seperti gedung minimalis tapi sangat efisien:

- Tidak banyak ornamen
- Fokus pada fungsi utama

- Mudah dirawat dan sangat stabil

Pada WhatsApp:

- Arsitektur sederhana sering kali lebih kuat
- Fokus pada kebutuhan inti pengguna
- Efisiensi lebih penting daripada kompleksitas

Software dengan arsitektur yang baik itu seperti bangunan yang tidak roboh saat ramai, mudah diperluas, tetap berfungsi walau sebagian rusak, dan nyaman digunakan dalam jangka panjang. Arsitektur yang baik tidak terlihat oleh pengguna, tetapi dirasakan dampaknya: cepat, stabil, dan jarang bermasalah.

B. Architectural Styles dan Design Patterns

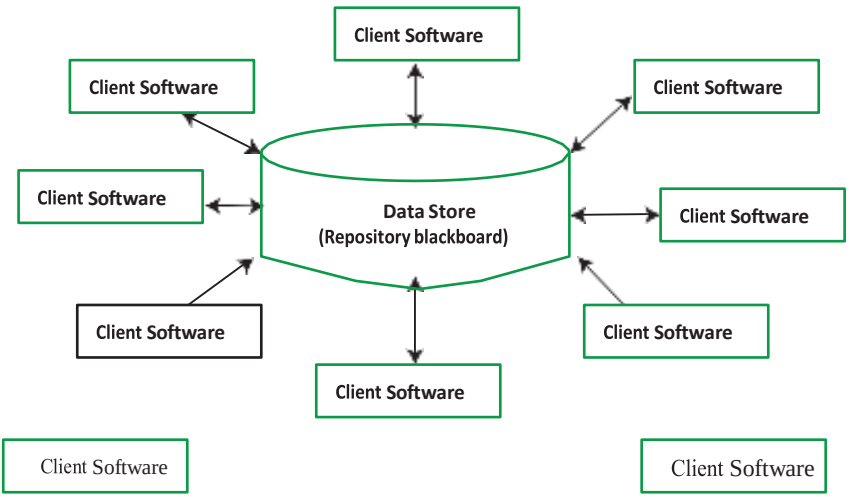
Dalam perancangan perangkat lunak modern, *architectural styles* dan *design patterns* berperan sebagai panduan utama untuk membangun sistem yang terstruktur dan mudah dikembangkan, meskipun keduanya berada pada tingkat abstraksi yang berbeda. *Architectural styles* menentukan bentuk besar sistem, yaitu bagaimana komponen utama disusun dan saling berinteraksi pada tingkat keseluruhan, seperti arsitektur berlapis, client–server, event-driven, atau service-based, yang masing-masing memiliki implikasi langsung terhadap kinerja, skalabilitas, dan kemudahan pemeliharaan. Sebaliknya, *design patterns* berfokus pada tingkat yang lebih rinci sebagai solusi umum terhadap masalah desain yang sering muncul di dalam komponen atau modul, seperti pengelolaan objek dan komunikasi antar bagian sistem. Hubungan keduanya bersifat saling melengkapi: gaya arsitektur menyediakan kerangka besar sistem, sementara *design patterns* memastikan kerangka tersebut diisi dengan

solusi desain yang konsisten dan teruji, sehingga sistem dapat diimplementasikan secara rapi, adaptif, dan berkelanjutan.

A. Gaya Arsitektur (Architectural Styles)

Gaya arsitektur mendefinisikan struktur tingkat tinggi dari sistem perangkat lunak, termasuk bagaimana komponen diorganisir dan cara mereka berkomunikasi. Pemilihan gaya ini menetapkan kerangka kerja untuk integrasi, komunikasi, dan perilaku sistem, sekaligus memberikan model semantik untuk memahami sifat keseluruhan sistem. Beberapa gaya arsitektur umum antara lain:

1. **Arsitektur Berpusat Data (Data-Centered Architecture)**



Gambar 1. Data-centered architecture

Dalam gaya ini, terdapat repositori data pusat yang menjadi inti sistem. Komponen (klien) mengakses repositori ini untuk

menambahkan, memperbarui, menghapus, atau mengambil data. Variasi termasuk sistem blackboard, di mana klien diberi notifikasi saat ada perubahan data yang relevan.

Keunggulan:

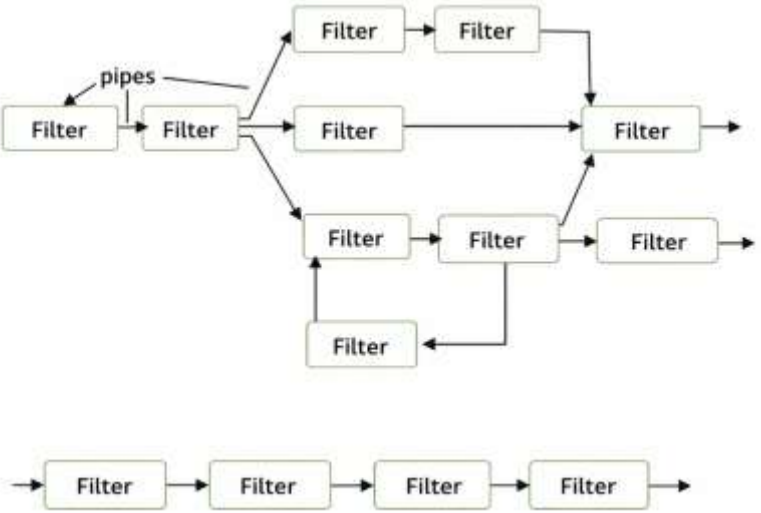
- Repositori data independen dari klien.
- Klien dapat bekerja secara independen.
- Menambahkan atau memodifikasi klien mudah dilakukan.

2. **Arsitektur Aliran Data (Data-Flow Architecture)**

Data masukan diubah menjadi data keluaran melalui serangkaian komponen komputasi yang disebut filter, yang dihubungkan oleh pipa. Setiap filter bekerja secara independen dan meneruskan data ke filter berikutnya. Jika aliran menjadi satu rantai sekuensial, disebut batch sequential.

Keunggulan: Mendukung pemeliharaan, penggunaan ulang, dan eksekusi paralel.

Kelemahan: Interaksi pengguna terbatas, koordinasi antara aliran paralel sulit dilakukan.

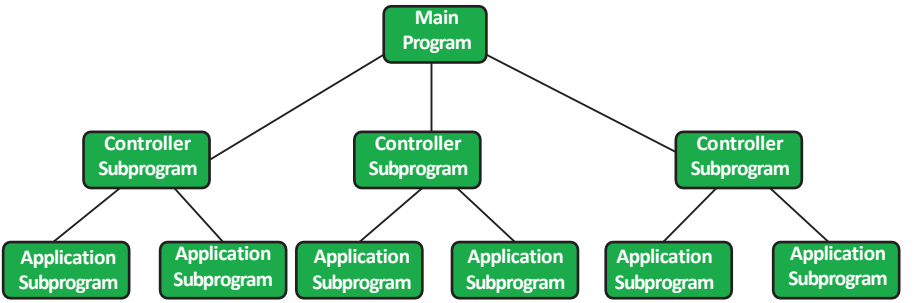


Gambar 2. Data-Flow Architecture

3. **Arsitektur Panggil-dan-Kembalikan (Call-and-Return Architecture)**

   Gaya ini mengorganisir program menjadi program utama dan subprogram, membentuk hierarki kontrol. Variannya termasuk arsitektur remote procedure call (RPC), di mana subprogram dapat dijalankan di beberapa komputer dalam jaringan.

   Keunggulan: Mudah diskalakan dan dimodifikasi.



Gambar 3. Call-and-Return Architecture

4. **Arsitektur Berorientasi Objek (Object-Oriented Architecture)**

   Komponen, atau objek, mengenkapsulasi data dan operasi yang memanipulasi data tersebut. Komunikasi antar objek terjadi melalui message passing, sehingga mendukung modularitas dan enkapsulasi.

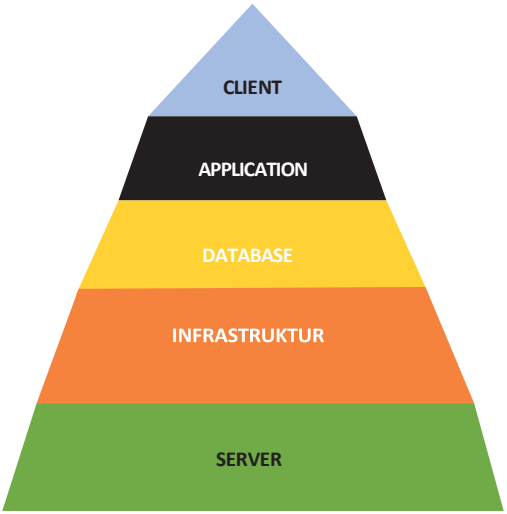
   Ciri dan Keunggulan:

   - Objek melindungi integritas sistem.
   - Objek tidak mengetahui detail internal objek lain, memungkinkan perubahan mandiri.
   - Memudahkan pemisahan masalah menjadi objek-objek mandiri.

5. **Arsitektur Berlapis (Layered Architecture)**

   Sistem dibagi menjadi lapisan hierarkis, masing-masing menangani operasi tertentu. Lapisan luar biasanya menangani antarmuka pengguna, lapisan tengah menangani logika

aplikasi, dan lapisan dalam berinteraksi dengan sistem operasi. Contoh terkenal: model OSI pada komunikasi jaringan. Keunggulan: Mendukung modularitas, pemeliharaan, dan abstraksi bertingkat ke operasi tingkat rendah.



Gambar 4. Layered Architecture

B. Pola Desain (Design Patterns)

Sementara gaya arsitektur menentukan struktur keseluruhan sistem, pola desain memberikan solusi yang dapat digunakan kembali untuk masalah yang sering muncul di dalam komponen. Pola desain bekerja pada tingkat abstraksi lebih rendah dibanding gaya arsitektur dan mengarahkan interaksi antara objek atau kelas.

Contoh Pola Desain Umum:

Pola	Tujuan	Contoh
Singleton	Memastikan hanya ada satu instance dari suatu kelas	Logger

Observer	Memberi notifikasi pada komponen lain saat terjadi perubahan	Event listener GUI
Factory	Mengenkapsulasi pembuatan objek	Pembuatan elemen UI
Strategy	Mengganti algoritma secara dinamis	Pemilihan metode sorting
Decorator	Menambah tanggung jawab secara dinamis	Membungkus stream input/output

Hubungan:

Sistem perangkat lunak biasanya menggunakan gaya arsitektur untuk struktur keseluruhan dan pola desain di dalamnya untuk menyelesaikan masalah implementasi dengan efisien.

C. Quality Attributes (Performance, Scalability, Reliability)

Dalam arsitektur perangkat lunak modern, keberhasilan sistem tidak hanya ditentukan oleh kelengkapan fitur, tetapi terutama oleh kualitas sistem dalam beroperasi pada kondisi nyata. Kualitas ini dikenal sebagai *quality attributes*, yaitu karakteristik non-fungsional yang menggambarkan bagaimana sistem merespons beban, gangguan, dan perubahan. Di antara berbagai atribut kualitas, kinerja (*performance*), skalabilitas (*scalability*), dan keandalan (*reliability*) merupakan aspek paling krusial dalam sistem skala besar karena secara langsung memengaruhi pengalaman pengguna dan keberlanjutan layanan.

Kinerja berkaitan dengan kecepatan dan responsivitas sistem dalam memproses permintaan, yang dirasakan pengguna melalui waktu respon, latensi, dan kelancaran interaksi. Kinerja sangat dipengaruhi oleh keputusan arsitektural, seperti pembagian komponen, alur komunikasi, dan pengelolaan sumber daya. Skalabilitas melengkapi kinerja dengan memastikan sistem mampu menangani

pertumbuhan beban tanpa penurunan kualitas layanan. Arsitektur yang mendukung skalabilitas biasanya bersifat modular dan memungkinkan penambahan kapasitas secara bertahap, sehingga sistem dapat tumbuh seiring peningkatan jumlah pengguna dan kebutuhan bisnis tanpa perubahan desain yang radikal.

Keandalan menunjukkan kemampuan sistem untuk tetap beroperasi secara konsisten serta menangani kegagalan secara terkendali. Dari sudut pandang arsitektur, keandalan dipengaruhi oleh isolasi komponen, mekanisme redundansi, dan strategi pemulihan kesalahan. Ketiga atribut ini saling berkaitan dan sering kali melibatkan *trade-off*, sehingga peningkatan satu atribut dapat berdampak pada atribut lainnya. Oleh karena itu, peran arsitektur adalah menyeimbangkan kinerja, skalabilitas, dan keandalan sesuai dengan konteks dan prioritas sistem, karena kualitas tersebut merupakan konsekuensi langsung dari keputusan desain sejak tahap awal pengembangan.

D. Trade-off dalam Perancangan Arsitektur

Dalam arsitektur perangkat lunak modern, setiap keputusan desain selalu melibatkan *trade-off*, yaitu kompromi antara satu atribut kualitas dengan atribut lainnya. Tidak ada arsitektur yang unggul di semua aspek sekaligus, sehingga arsitek harus menetapkan prioritas sesuai tujuan dan konteks sistem. Berikut trade-off utama yang umum muncul pada sistem skala besar.

1. **Kinerja (Performance) vs Keandalan (Reliability)**. Optimasi kinerja menuntut jalur pemrosesan yang ringkas dan minim lapisan, tetapi dapat mengurangi mekanisme pemulihan. Sebaliknya, peningkatan keandalan melalui

redundansi dan pengecekan kesalahan sering menambah latensi.

2. **Skalabilitas vs Kompleksitas Sistem**. Skalabilitas tinggi dicapai dengan memecah sistem menjadi banyak komponen yang mandiri, namun hal ini meningkatkan kompleksitas pengelolaan, komunikasi, dan pemantauan. Sistem sederhana lebih mudah dikelola, tetapi cepat mencapai batas kapasitas.

3. **Konsistensi Data vs Ketersediaan Layanan**. Menjaga konsistensi data yang ketat dapat menurunkan ketersediaan saat terjadi gangguan. Sebaliknya, memprioritaskan ketersediaan memungkinkan layanan tetap berjalan meskipun data belum sepenuhnya sinkron.

4. **Fleksibilitas Arsitektur vs Biaya Pengembangan**. Arsitektur yang fleksibel memerlukan investasi awal lebih besar, tetapi memudahkan perubahan di masa depan. Arsitektur sederhana lebih murah di awal, namun berpotensi mahal saat sistem berkembang.

5. **Standarisasi vs Kecepatan Inovasi**. Standarisasi meningkatkan konsistensi dan kolaborasi, tetapi dapat membatasi inovasi. Kebebasan desain mempercepat inovasi jangka pendek, namun berisiko menurunkan keterkelolaan jangka panjang.

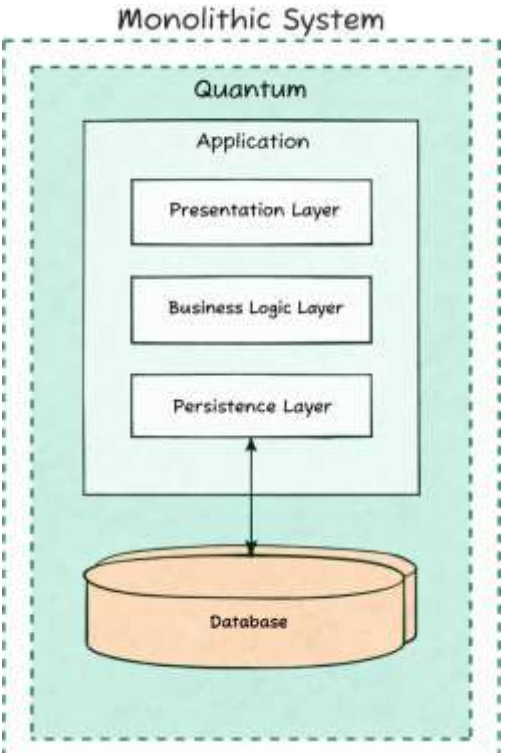
6. **Otomatisasi vs Kontrol Manual**. Otomatisasi meningkatkan efisiensi dan konsistensi, tetapi mengurangi kontrol langsung. Kontrol manual lebih hati-hati, namun kurang efektif untuk sistem besar dan dinamis.

Pemahaman trade-off ini membantu arsitek membuat keputusan yang realistis dan kontekstual. Keberhasilan arsitektur bukan menghilangkan trade-off, melainkan memilih kompromi yang paling tepat sesuai tujuan sistem dan kebutuhan jangka panjang organisasi.

E. Arsitektur Berlapis dan Modularitas

Arsitektur berlapis (*layered architecture*) merupakan salah satu pendekatan paling mendasar dan banyak digunakan dalam perancangan perangkat lunak modern untuk mengelola kompleksitas sistem. Pendekatan ini menyusun sistem ke dalam lapisan-lapisan dengan tanggung jawab yang jelas, sehingga setiap bagian sistem dapat dikembangkan, diuji, dan dipelihara secara lebih terkontrol. Konsep ini sangat erat kaitannya dengan modularitas, yaitu prinsip pemecahan sistem ke dalam bagian-bagian yang relatif mandiri dan tidak saling bercampur tanggung jawabnya.

Struktur arsitektur berlapis dapat dilihat secara jelas pada gambar, yang menggambarkan sebuah sistem monolitik modular. Pada gambar tersebut, sistem dibagi ke dalam beberapa lapisan utama, yaitu *Presentation Layer*, *Business Logic Layer*, dan *Persistence Layer*, yang kemudian terhubung ke *Database*. Setiap lapisan memiliki peran yang spesifik. *Presentation Layer* bertanggung jawab terhadap interaksi dengan pengguna, *Business Logic Layer* menangani aturan dan proses bisnis, sedangkan *Persistence Layer* mengelola akses dan penyimpanan data. Pemisahan ini menunjukkan bagaimana modularitas dapat diterapkan secara konseptual, bahkan dalam sistem yang masih bersifat monolitik.



Gambar 5. Arsitektur berlapis pada sistem monolitik modular

Melalui gambar, terlihat bahwa arsitektur berlapis mengatur arah ketergantungan antar komponen secara jelas. Lapisan atas tidak berinteraksi langsung dengan lapisan bawah tanpa melalui mekanisme yang telah ditentukan. Sebagai contoh, lapisan presentasi tidak mengakses basis data secara langsung, tetapi melalui lapisan logika bisnis dan persistensi. Pola ini mencegah ketergantungan yang tidak terkontrol serta membatasi dampak perubahan. Ketika teknologi penyimpanan data berubah, lapisan lain tidak perlu dimodifikasi secara signifikan selama kontrak antarlapisan tetap dijaga (Sommerville, 2016; Bass, Clements, & Kazman, 2022).

Modularitas yang ditunjukkan pada gambar memperlihatkan bagaimana sistem dapat dikembangkan secara bertahap melalui pembagian tanggung jawab yang jelas. Setiap lapisan berfungsi sebagai modul mandiri dengan tingkat kohesi tinggi dan ketergantungan minimal terhadap lapisan lain, sejalan dengan prinsip *high cohesion* dan *low coupling*. Struktur ini mempermudah pengujian, pelacakan kesalahan, dan kolaborasi tim secara paralel. Namun, arsitektur berlapis tetap perlu dirancang secara proporsional, karena jumlah lapisan yang berlebihan atau interaksi yang terlalu kaku dapat menurunkan kinerja dan fleksibilitas sistem. Oleh karena itu, arsitektur berlapis berfungsi sebagai fondasi pengendalian kompleksitas yang efektif sekaligus titik awal menuju evolusi arsitektur yang lebih adaptif (Pressman & Maxim, 2020).

F. Evaluasi Arsitektur Sistem

Evaluasi arsitektur sistem merupakan proses penting untuk memastikan bahwa rancangan arsitektur benar-benar mendukung tujuan sistem, baik dari sisi fungsional maupun kualitas. Arsitektur tidak bersifat final, melainkan keputusan strategis yang perlu ditinjau dan disempurnakan secara berkala. Melalui evaluasi, tim dapat mengidentifikasi risiko sejak dini, seperti ketergantungan komponen yang berlebihan, bottleneck kinerja, atau keterbatasan skalabilitas, yang sering kali baru muncul ketika sistem berkembang dan menerima beban nyata. Deteksi awal ini membantu mencegah biaya perbaikan yang jauh lebih besar di tahap lanjutan.

Selain menilai aspek teknis seperti kinerja, skalabilitas, keandalan, dan keamanan, evaluasi arsitektur juga memiliki peran organisasional. Arsitektur berfungsi sebagai sarana komunikasi bersama antara pengembang, manajer, dan pemangku kepentingan

non-teknis. Proses evaluasi memastikan keselarasan pemahaman terhadap struktur sistem, trade-off desain, dan arah pengembangan jangka panjang. Dalam praktik modern, evaluasi dilakukan secara berulang seiring perubahan fitur, teknologi, dan skala sistem, sehingga arsitektur tetap adaptif dan relevan. Evaluasi bukanlah tanda kegagalan, melainkan bagian dari peningkatan berkelanjutan dalam rekayasa perangkat lunak.

.

BAB III
ARSITEKTUR BERBASIS LAYANAN (SERVICE-ORIENTED & MICROSERVICES)

Perkembangan sistem perangkat lunak modern mendorong perubahan mendasar dalam cara aplikasi dirancang dan dioperasikan. Ketika kebutuhan bisnis menuntut kecepatan inovasi, ketersediaan tinggi, dan kemampuan beradaptasi terhadap skala pengguna yang terus berubah, pendekatan arsitektur tradisional mulai menunjukkan keterbatasannya. Dalam konteks inilah arsitektur berbasis layanan muncul sebagai jawaban atas kompleksitas sistem berskala besar. Dengan memecah sistem menjadi layanan-layanan yang lebih kecil dan mandiri, organisasi dapat mengembangkan, menguji, dan memelihara perangkat lunak secara lebih fleksibel dan terkontrol.

Bab ini membahas dua pendekatan utama dalam arsitektur berbasis layanan, yaitu *Service-Oriented Architecture* (SOA) dan *microservices*. Pembahasan dimulai dari konsep dasar dan prinsip desain layanan, kemudian berkembang ke perbandingan antara arsitektur monolitik dan microservices, mekanisme komunikasi antar layanan, serta peran komponen pendukung seperti API gateway dan service discovery. Selain menyoroti manfaat skalabilitas dan ketahanan sistem, bab ini juga mengulas tantangan nyata dalam implementasi microservices, sehingga pembaca memperoleh pemahaman yang seimbang antara potensi dan risiko arsitektur berbasis layanan dalam pengembangan sistem perangkat lunak modern.

A. Konsep Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) adalah pendekatan arsitektur perangkat lunak yang memecah sistem menjadi layanan-layanan mandiri yang merepresentasikan fungsi bisnis tertentu dan dapat digunakan kembali. Setiap layanan berinteraksi melalui antarmuka yang terdefinisi jelas, sehingga sistem menjadi lebih mudah dikembangkan, dipelihara, dan diintegrasikan dibandingkan pendekatan monolitik, terutama ketika skala dan kompleksitas meningkat.

Inti SOA terletak pada prinsip loose coupling, di mana ketergantungan antar layanan dibuat seminimal mungkin. Selama kontrak antarmuka tetap konsisten, perubahan pada satu layanan tidak secara langsung memengaruhi layanan lain. Prinsip ini memungkinkan sistem berkembang secara bertahap dan adaptif terhadap perubahan kebutuhan bisnis, khususnya dalam lingkungan organisasi besar.

SOA juga menekankan interoperabilitas, memungkinkan layanan berkomunikasi lintas platform, bahasa pemrograman, dan sistem yang berbeda. Pendekatan ini banyak digunakan untuk mengintegrasikan sistem lama (*legacy systems*) dengan aplikasi baru tanpa harus membangun ulang seluruh sistem, sehingga membantu organisasi memaksimalkan investasi teknologi yang sudah ada.

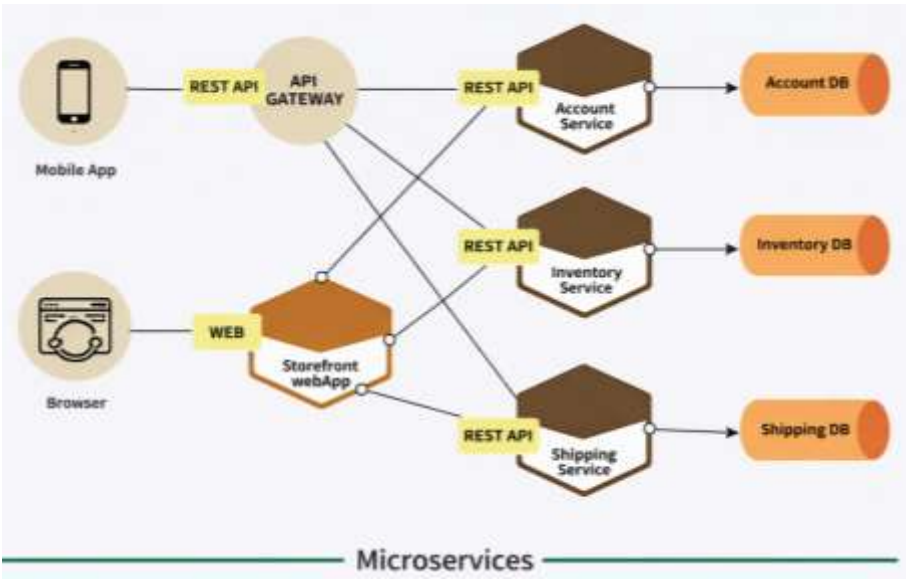
Dari perspektif bisnis, SOA berfungsi sebagai jembatan antara proses bisnis dan implementasi teknis. Setiap layanan biasanya selaras dengan fungsi bisnis yang jelas, sehingga perubahan proses bisnis dapat diterjemahkan ke dalam penyesuaian layanan tertentu tanpa mengganggu keseluruhan sistem. Namun, penerapan SOA menuntut

governance yang baik, termasuk pengelolaan versi, keamanan, dan pemantauan kinerja, agar kompleksitas sistem tetap terkendali.

SOA dapat dipandang sebagai fondasi awal arsitektur berbasis layanan modern. Banyak prinsipnya menjadi dasar berkembangnya microservices, meskipun SOA lebih menekankan integrasi dan standardisasi enterprise, sementara microservices berfokus pada layanan yang lebih kecil dan otonom. Pemahaman SOA penting untuk memahami evolusi arsitektur berbasis layanan secara menyeluruh.

B. Arsitektur Microservices

Arsitektur microservices merupakan pendekatan pengembangan perangkat lunak yang memecah sistem menjadi layanan-layanan kecil, mandiri, dan berfokus pada satu fungsi bisnis yang spesifik. Setiap layanan dikembangkan dan dideploy secara independen serta berkomunikasi melalui antarmuka yang ringan. Pendekatan ini berkembang dari Service-Oriented Architecture (SOA) dengan tujuan meningkatkan fleksibilitas, mempercepat pengembangan, dan memudahkan adaptasi terhadap perubahan kebutuhan.



Gambar 6. Arsitektur microservices dengan API gateway dan basis data terpisah

Konsep ini dapat dipahami melalui Gambar 6, yang menampilkan arsitektur microservices dengan API Gateway sebagai pintu masuk utama sistem. Klien, seperti aplikasi web dan mobile, tidak berinteraksi langsung dengan seluruh layanan backend, melainkan melalui gateway yang mengatur routing, keamanan, dan kontrol akses. Pola ini menyederhanakan interaksi klien sekaligus meningkatkan konsistensi dan keamanan sistem.

Ciri penting microservices yang tampak pada Gambar 6 adalah otonomi layanan dan pemisahan data. Setiap layanan, seperti *Account Service*, *Inventory Service*, dan *Shipping Service*, memiliki basis data tersendiri. Pemisahan ini memungkinkan setiap layanan berkembang

secara independen dan mengurangi dampak kegagalan antar layanan, sehingga meningkatkan ketahanan sistem.

Selain itu, microservices mendukung skalabilitas granular, di mana hanya layanan dengan beban tinggi yang perlu diskalakan tanpa memengaruhi keseluruhan sistem. Namun, arsitektur ini juga menimbulkan tantangan operasional, seperti latensi jaringan, kompleksitas pengelolaan layanan, dan kebutuhan monitoring serta otomasi yang matang.

Secara keseluruhan, microservices menawarkan fleksibilitas dan skalabilitas tinggi, tetapi menuntut kesiapan teknis dan organisasi yang memadai. Visualisasi pada Gambar 6 menegaskan bahwa microservices bukan sekadar pemecahan sistem, melainkan pendekatan arsitektural menyeluruh yang mencakup desain layanan, pengelolaan data, dan pola komunikasi.

C. Perbandingan Monolithic vs Microservices

Arsitektur monolithic dan microservices merepresentasikan dua pendekatan berbeda dalam membangun sistem perangkat lunak. Monolithic mengintegrasikan seluruh komponen sistem—antarmuka, logika bisnis, dan akses data—ke dalam satu aplikasi terpadu. Sebaliknya, microservices memecah sistem menjadi layanan-layanan kecil yang berdiri mandiri dan saling berkomunikasi melalui jaringan. Perbedaan ini berpengaruh langsung terhadap pengembangan, pengelolaan, dan skalabilitas sistem.

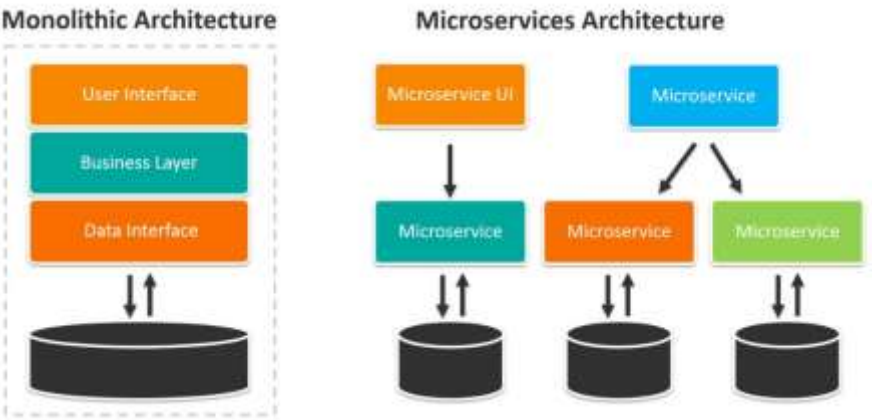
Dari sisi pengembangan, monolithic lebih sederhana pada tahap awal. Satu basis kode memudahkan koordinasi tim kecil serta proses pengujian dan deployment. Namun, seiring sistem membesar, kompleksitas meningkat dan perubahan kecil berisiko memengaruhi

keseluruhan aplikasi. Kondisi ini membuat monolithic kurang fleksibel untuk sistem yang berkembang cepat.

Microservices menawarkan fleksibilitas lebih tinggi dengan memungkinkan setiap layanan dikembangkan dan dideploy secara independen. Pendekatan ini mendukung kerja paralel tim dan inovasi berkelanjutan. Di sisi lain, kompleksitas operasional meningkat karena banyaknya layanan, kebutuhan komunikasi jaringan, serta tuntutan monitoring dan otomatisasi yang lebih matang.

Dari perspektif skalabilitas dan keandalan, monolithic diskalakan secara menyeluruh sehingga kurang efisien ketika hanya sebagian fungsi yang mengalami lonjakan beban. Microservices memungkinkan skalabilitas granular dan isolasi kegagalan, sehingga satu layanan bermasalah tidak selalu menghentikan sistem secara keseluruhan.

Pemilihan arsitektur sangat bergantung pada konteks dan kesiapan organisasi. Monolithic cocok untuk sistem kecil atau tahap awal pengembangan, sedangkan microservices lebih sesuai untuk sistem berskala besar dengan kebutuhan fleksibilitas dan pertumbuhan jangka panjang.



Gambar 7. Perbandingan arsitektur monolitik dan arsitektur microservices

Perbedaan antara arsitektur monolitik dan microservices dapat dilihat secara visual pada Gambar di atas. Pada sisi monolitik, seluruh komponen sistem antarmuka pengguna, logika bisnis, dan akses data tergabung dalam satu kesatuan aplikasi yang terhubung ke satu basis data terpusat. Struktur ini membuat sistem relatif mudah dikembangkan pada tahap awal, tetapi menjadi semakin sulit dipelihara dan diskalakan ketika kompleksitas meningkat. Sebaliknya, sisi microservices pada Gambar di atas menunjukkan sistem yang dipecah ke dalam layanan-layanan kecil dan mandiri, masing-masing dengan basis data sendiri. Pendekatan ini memungkinkan pengembangan, deployment, dan skalabilitas dilakukan secara independen, namun menuntut pengelolaan sistem yang lebih matang.

Dengan demikian, tidak ada pendekatan yang secara mutlak lebih unggul. Monolithic dan microservices masing-masing memiliki konteks penggunaan yang ideal. Monolithic cocok untuk sistem sederhana, tim kecil, dan kebutuhan yang relatif stabil. Microservices lebih sesuai untuk sistem skala besar, kebutuhan perubahan cepat, dan

organisasi dengan kemampuan teknis yang matang. Perbandingan ini menegaskan bahwa pemilihan arsitektur harus didasarkan pada kebutuhan sistem dan kesiapan organisasi, bukan semata-mata mengikuti tren teknologi.

D. Komunikasi Antar Layanan (REST, gRPC, Event-Driven)

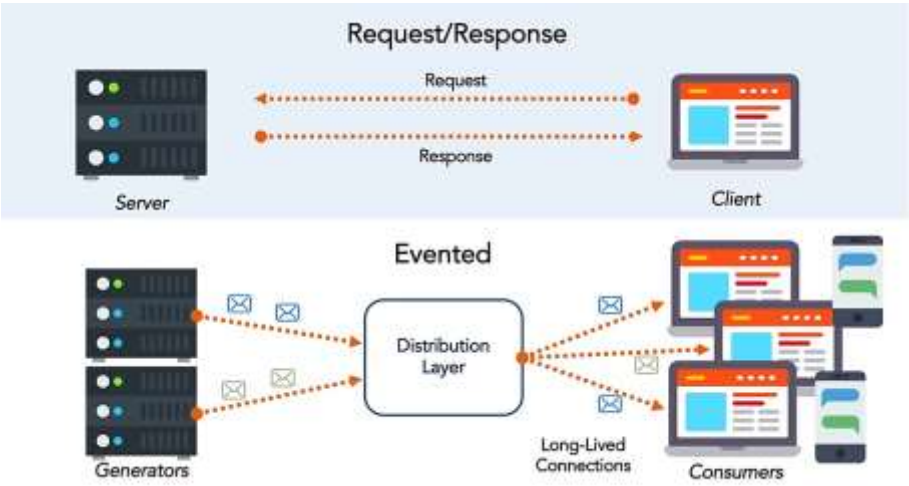
Dalam arsitektur microservices, komunikasi antar layanan menjadi keputusan arsitektural penting karena berlangsung melalui jaringan dan secara langsung memengaruhi kinerja, skalabilitas, serta keandalan sistem. Pemilihan pola komunikasi sinkron atau asinkron menentukan bagaimana sistem merespons beban dan kegagalan layanan (Bass, Clements, & Kazman, 2022).

REST merupakan pendekatan sinkron yang paling umum, menggunakan pola request–response. REST mudah dipahami dan cocok untuk interaksi langsung yang membutuhkan respons cepat, seperti autentikasi, pencarian data, atau pengecekan status. Namun, sifat sinkron membuat layanan pemanggil bergantung pada ketersediaan layanan tujuan, sehingga berpotensi menurunkan kinerja saat terjadi latensi atau kegagalan (Richardson et al., 2013).

Untuk kebutuhan performa lebih tinggi, gRPC digunakan sebagai alternatif sinkron berbasis protokol biner. gRPC menawarkan latensi rendah dan efisiensi pertukaran data, sehingga sesuai untuk komunikasi internal antar layanan. Meski lebih cepat dibanding REST, gRPC tetap membawa risiko ketergantungan antar layanan yang melekat pada komunikasi sinkron (Newman, 2021).

Pendekatan event-driven bersifat asinkron dan berbasis peristiwa. Layanan mempublikasikan event tanpa memanggil layanan lain secara langsung, memungkinkan layanan lain bereaksi secara mandiri. Pola

ini meningkatkan loose coupling dan ketahanan sistem, tetapi menambah kompleksitas dalam pemantauan alur proses dan konsistensi data (Newman, 2021).



Gambar 8. Perbandingan komunikasi sinkron (REST) dan asinkron (event-driven) dalam arsitektur layanan

Perbedaan pendekatan komunikasi antar layanan dapat dilihat pada Gambar. Pada komunikasi berbasis REST, interaksi antara *consumer* dan *provider* bersifat sinkron melalui pola *request/response*, di mana layanan pemanggil harus menunggu respon sebelum melanjutkan proses. Sebaliknya, pendekatan event-driven pada Gambar menunjukkan komunikasi asinkron, di mana layanan berlangganan terhadap suatu peristiwa dan menerima notifikasi ketika peristiwa tersebut terjadi. Pola ini mengurangi ketergantungan langsung antar layanan dan meningkatkan fleksibilitas serta ketahanan sistem terdistribusi.

Dalam praktik arsitektur modern, REST, gRPC, dan event-driven tidak saling menggantikan, melainkan saling melengkapi. REST atau gRPC umumnya digunakan untuk interaksi langsung yang membutuhkan respon cepat, sementara event-driven digunakan untuk proses bisnis yang bersifat reaktif dan berjalan di belakang layar. Pemilihan dan kombinasi pendekatan komunikasi ini harus disesuaikan dengan kebutuhan sistem, karakteristik beban kerja, serta tingkat kompleksitas yang dapat dikelola oleh organisasi.

E. Service Discovery dan API Gateway

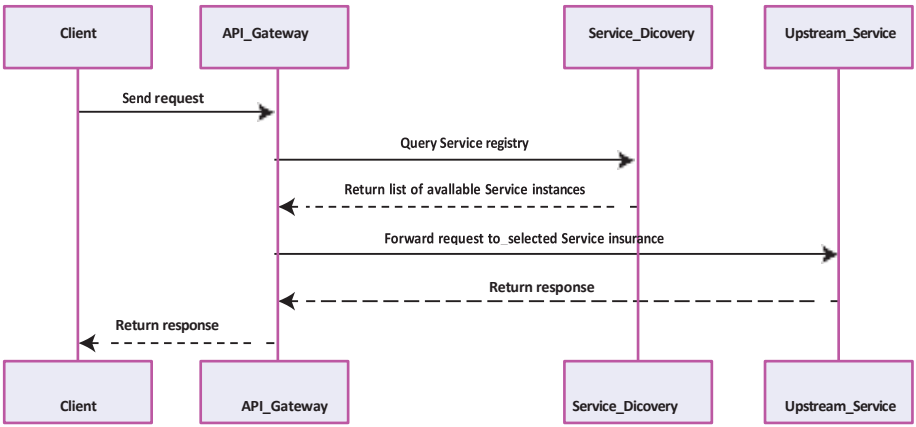
Dalam arsitektur microservices, layanan bersifat dinamis: instans dapat bertambah, berkurang, atau berpindah lokasi akibat auto-scaling dan orkestrasi. Kondisi ini membuat alamat layanan tidak dapat ditentukan secara statis sebagaimana pada sistem monolitik. Untuk mengelola dinamika tersebut, dua komponen arsitektural yang krusial adalah *service discovery* dan *API gateway* (Bass et al., 2022).

Service discovery memungkinkan layanan saling menemukan secara otomatis. Setiap layanan yang aktif mendaftarkan dirinya ke *service registry* dengan identitas tertentu. Ketika layanan lain membutuhkan akses, ia cukup meminta lokasi layanan tersebut ke registry, bukan ke alamat tetap. Mekanisme ini membuat komunikasi antar layanan tetap berjalan meskipun terjadi perubahan jumlah instans atau lokasi deployment, sehingga meningkatkan fleksibilitas dan ketahanan sistem (Newman, 2021).

Berbeda dari *service discovery* yang berfokus pada komunikasi internal, *API gateway* berfungsi sebagai pintu masuk tunggal bagi klien eksternal. Klien tidak perlu mengetahui struktur maupun alamat layanan backend, karena seluruh permintaan diarahkan melalui

gateway. API gateway menangani routing, penggabungan respons, serta aspek keamanan, sehingga kompleksitas di sisi klien dapat ditekan dan perubahan arsitektur backend tidak berdampak langsung pada pengguna (Richardson, 2018).

Dalam sistem modern, *service discovery* dan *API gateway* bekerja secara terpadu. API gateway memanfaatkan *service discovery* untuk menemukan layanan backend yang aktif saat meneruskan permintaan klien. Kombinasi ini memungkinkan sistem berskala besar tetap adaptif, konsisten, dan mudah dikelola meskipun terdiri dari banyak layanan yang berubah secara dinamis.



Gambar 9. Alur Kerja API Gateway dan Service Discovery dalam Arsitektur Microservices
api7.ai

Gambar tersebut memperlihatkan alur komunikasi nyata antara *Client*, *API Gateway*, *Service Discovery*, dan *Upstream Service* dalam sistem microservices. Diagram ini menjelaskan bagaimana sebuah

permintaan pengguna diproses secara dinamis tanpa ketergantungan pada alamat layanan yang statis.

Langkah 1 – Client Mengirim Permintaan

Proses dimulai ketika Client (misalnya aplikasi mobile atau web) mengirimkan permintaan ke API Gateway. Client tidak berkomunikasi langsung dengan layanan backend. Hal ini menjaga agar struktur internal sistem tetap tersembunyi dan aman.

Contoh konkret: Aplikasi mobile mengirim permintaan *"ambil status pesanan"* ke API Gateway.

Langkah 2 – API Gateway Menghubungi Service Discovery

Setelah menerima permintaan, API Gateway tidak langsung meneruskannya ke layanan backend. Sebaliknya, gateway terlebih dahulu melakukan query ke Service Discovery untuk mencari tahu:

"Instans layanan mana yang saat ini aktif dan tersedia?"

Service Discovery berfungsi sebagai direktori layanan dinamis yang selalu diperbarui.

Langkah 3 – Service Discovery Mengembalikan Daftar Layanan Aktif

Service Discovery mengembalikan daftar instans layanan yang tersedia, misalnya beberapa instans *Order Service* yang sedang berjalan.

Pada tahap ini:

- Gateway tidak terikat pada satu alamat tetap

- Gateway dapat memilih instans yang paling sesuai (misalnya paling ringan bebannya)

Ini memungkinkan load balancing dan auto-scaling berjalan secara efektif.

Langkah 4 – API Gateway Meneruskan Permintaan

Setelah memilih instans yang tepat, API Gateway meneruskan permintaan ke Upstream Service yang dipilih.

Upstream Service kemudian memproses permintaan tersebut, misalnya mengambil data dari basis data atau memanggil layanan lain di belakang layar.

Langkah 5 – Respons Dikembalikan ke Client

Respons dari layanan backend dikirim kembali ke **API Gateway**, lalu diteruskan ke **Client** sebagai hasil akhir.

Dari sudut pandang Client:

- Proses terlihat sederhana
- Client hanya berkomunikasi dengan satu endpoint
- Tidak mengetahui kompleksitas internal sistem

Makna Arsitektural dari Gambar

Diagram ini menegaskan beberapa prinsip penting microservices:

1. **Loose coupling**. Client dan layanan backend tidak saling bergantung langsung.
2. **Dynamic service resolution**. Alamat layanan ditentukan saat runtime, bukan secara statis.

3. **Scalability by design**. Layanan dapat bertambah atau berkurang tanpa mengganggu client.
4. **Centralized control**. API Gateway menjadi titik pengelolaan keamanan, trafik, dan observability.

Keberhasilan microservices sangat bergantung pada desain dan penerapan service discovery serta API gateway yang tepat. Pemahaman ini menjadi landasan penting sebelum membahas tantangan implementasi microservices secara lebih kritis pada subbab berikutnya.

F. Tantangan Implementasi Microservices

Meskipun arsitektur microservices menawarkan fleksibilitas, skalabilitas, dan ketahanan sistem yang tinggi, implementasinya menghadirkan kompleksitas signifikan. Perubahan dari pemanggilan internal ke komunikasi jaringan meningkatkan risiko latensi, kegagalan parsial, dan kesulitan debugging. Pengelolaan data menjadi lebih rumit karena setiap layanan memiliki basis data sendiri, sehingga konsistensi lintas layanan sulit dijaga dan sering kali memerlukan penerimaan model *eventual consistency*. Selain itu, observability menjadi tantangan utama karena log, metrik, dan jejak eksekusi tersebar di banyak layanan.

Di sisi operasional dan organisasi, microservices sangat bergantung pada infrastruktur otomatis, praktik DevOps yang matang, serta komponen pendukung seperti API gateway dan service discovery yang harus dirancang tahan gagal. Permukaan serangan yang lebih luas menuntut mekanisme keamanan antar layanan yang ketat. Penerapan yang tidak tepat juga berisiko menimbulkan *over-engineering*, peningkatan biaya operasional, serta kegagalan

koordinasi tim. Oleh karena itu, microservices harus dipahami sebagai strategi arsitektural jangka panjang yang memerlukan kesiapan teknis dan organisasi, bukan sekadar solusi teknis instan.

BAB IV
ARSITEKTUR CLOUD-NATIVE DAN CONTAINERIZATION

Transformasi digital dalam satu dekade terakhir telah mengubah cara perangkat lunak dibangun, dijalankan, dan dikelola, karena infrastruktur tradisional berbasis server fisik dan konfigurasi statis makin tidak memadai untuk memenuhi tuntutan sistem yang harus selalu tersedia, skalabel, dan adaptif terhadap perubahan bisnis. Dalam situasi ini, komputasi awan menjadi fondasi utama yang memungkinkan akses sumber daya secara elastis dan berbasis layanan, sekaligus membentuk paradigma baru dalam rekayasa perangkat lunak. Bab ini membahas arsitektur cloud-native dan containerization sebagai pendekatan modern dalam pengembangan sistem, mencakup peran cloud, prinsip aplikasi cloud-native, penggunaan container dan virtualisasi, orkestrasi container untuk sistem skala besar, praktik CI/CD sebagai inti pengembangan modern, serta strategi multi-cloud dan hybrid cloud, guna memberikan landasan konseptual dan praktis tentang bagaimana perangkat lunak dirancang agar fleksibel, tangguh, dan siap beroperasi dalam lingkungan cloud yang dinamis.

A. Komputasi Awan dalam Rekayasa Perangkat Lunak

Cloud computing adalah pendekatan komputasi yang memungkinkan pengguna mengakses sumber daya teknologi seperti komputasi, penyimpanan data, dan layanan perangkat lunak melalui jaringan, umumnya internet. Dalam model ini, organisasi tidak perlu lagi memiliki dan mengelola server fisik sendiri di lokasi internal. Sebaliknya, mereka menggunakan sumber daya dari pusat data (*data*

center) milik penyedia layanan cloud yang menyediakan akses ke server-server tersebut secara terukur dan fleksibel.

Secara sederhana, "cloud" dapat dipahami sebagai kumpulan pusat data yang berisi banyak server. Ketika data disimpan di cloud, data tersebut berada pada server khusus yang dirancang untuk penyimpanan dan pengolahan skala besar. Demikian pula, aplikasi yang dijalankan di cloud memanfaatkan daya komputasi dari satu atau lebih server yang tersedia di pusat data tersebut.

Cloud Computing dalam Praktik Rekayasa Perangkat Lunak

Bagi pengembang perangkat lunak, cloud computing menyediakan infrastruktur terdistribusi jarak jauh yang dapat disesuaikan dengan kebutuhan proyek. Lingkungan pengembangan, pengujian, dan produksi dapat dibuat dengan cepat tanpa harus membeli perangkat keras baru. Infrastruktur ini juga mudah diubah, baik untuk menambah kapasitas maupun mengurangi sumber daya yang tidak lagi dibutuhkan.

Penyedia layanan cloud biasanya memiliki pusat data di berbagai wilayah geografis. Hal ini memungkinkan aplikasi yang dibangun di cloud memiliki akses global dengan waktu respons yang relatif cepat, terlepas dari lokasi pengguna. Fleksibilitas ini menjadi faktor penting dalam pengembangan perangkat lunak modern yang melayani pengguna dalam skala besar dan lintas wilayah.

Struktur Aplikasi Berbasis Cloud

Aplikasi berbasis cloud umumnya dibagi menjadi dua bagian utama:

1. **Bagian server-side**, yang dijalankan di server cloud dalam pusat data. Bagian ini menangani logika bisnis, pemrosesan data, dan integrasi dengan sistem lain.

2. **Bagian client-side**, yang berjalan di perangkat pengguna. Bagian ini dapat berupa aplikasi yang diinstal melalui *application store* (seperti Android atau iOS) atau aplikasi berbasis web yang diakses melalui browser tanpa instalasi tambahan.

Dengan struktur ini, sebagian besar pemrosesan dilakukan di cloud, sehingga perangkat pengguna tidak memerlukan spesifikasi perangkat keras yang tinggi.

Jenis Cloud Berdasarkan Kepemilikan

Berdasarkan kepemilikan dan kebijakan akses, cloud dibedakan menjadi:

- **Public Cloud**, yaitu cloud yang dikelola oleh penyedia layanan dan digunakan oleh banyak organisasi.

- **Private Cloud**, yaitu cloud yang digunakan secara eksklusif oleh satu organisasi. Model ini memberikan kontrol dan keamanan lebih tinggi, tetapi memerlukan biaya dan pengelolaan yang besar.

- **Hybrid Cloud**, yaitu kombinasi antara public dan private cloud, yang memungkinkan organisasi menempatkan data sensitif di lingkungan tertutup dan beban kerja lain di cloud publik.

Private cloud sering digunakan pada sektor yang menuntut tingkat keamanan tinggi, seperti kesehatan, keuangan digital, dan penelitian.

Model Layanan Cloud

Penyedia cloud umumnya menawarkan tiga model layanan utama:

1. Infrastructure as a Service (IaaS)

IaaS menyediakan infrastruktur dasar seperti mesin virtual, penyimpanan, jaringan, dan *load balancer*. Pengguna memiliki kontrol besar atas sistem operasi dan aplikasi, tetapi juga bertanggung jawab atas pemeliharaannya. Model ini menyerupai "lahan kosong digital" tempat pengembang membangun sistem sesuai kebutuhan.

2. Platform as a Service (PaaS)

PaaS menyediakan lingkungan siap pakai untuk menjalankan aplikasi, termasuk sistem operasi, *runtime*, basis data, dan web server. Dengan model ini, pengembang dapat fokus pada pengembangan aplikasi tanpa harus mengelola infrastruktur secara detail.

3. Software as a Service (SaaS)

SaaS memungkinkan pengguna langsung menggunakan perangkat lunak yang dihosting di cloud melalui browser. Aplikasi SaaS biasanya berbasis langganan dan dikelola sepenuhnya oleh penyedia layanan, termasuk pembaruan dan pemeliharaan sistem.

Manfaat Cloud Computing dalam Rekayasa Perangkat Lunak

Penggunaan cloud computing memberikan berbagai manfaat utama, antara lain:

- Efisiensi biaya karena tidak memerlukan investasi perangkat keras besar
- Kemudahan kolaborasi tim lintas lokasi
- Jangkauan pengguna global
- Skalabilitas yang fleksibel
- Pengurangan beban pemeliharaan infrastruktur
- Dukungan keamanan dan pemulihan bencana
- Peningkatan produktivitas pengembangan perangkat lunak

Tantangan dan Risiko Cloud Computing

Di balik manfaatnya, cloud computing juga memiliki risiko, antara lain:

- Isu keamanan akibat sentralisasi data
- Ketergantungan pada penyedia layanan dan koneksi internet
- Potensi gangguan layanan yang berdampak luas pada operasional organisasi

Dengan segala kelebihan dan tantangannya, cloud computing telah menjadi fondasi utama rekayasa perangkat lunak modern. Pemahaman terhadap konsep, model layanan, serta implikasi cloud computing sangat penting sebelum melanjutkan ke pembahasan aplikasi cloud-native dan teknologi pendukung seperti container dan orkestrasi pada subbab berikutnya.

B. Konsep Cloud-Native Application

Banyak aplikasi awalnya hanya dipindahkan dari server lokal ke cloud tanpa perubahan desain yang signifikan. Pendekatan ini membuat aplikasi memang berjalan di cloud, tetapi tidak memanfaatkan sepenuhnya keunggulan komputasi awan. Keterbatasan ini mendorong lahirnya konsep *cloud-native application*, yaitu pendekatan pengembangan perangkat lunak yang sejak awal dirancang untuk lingkungan cloud yang dinamis dan terdistribusi.

Cloud-native hadir sebagai jawaban atas kebutuhan sistem yang harus selalu tersedia, mudah diskalakan, dan cepat beradaptasi terhadap perubahan beban kerja serta kebutuhan bisnis.

Cloud-native application adalah aplikasi yang dirancang, dikembangkan, dan dijalankan dengan memanfaatkan karakteristik utama cloud computing. Aplikasi ini tidak bergantung pada server tertentu dan menganggap infrastruktur sebagai sumber daya yang dapat berubah sewaktu-waktu.

Berbeda dengan aplikasi tradisional, cloud-native application dirancang dengan asumsi bahwa kegagalan dapat terjadi kapan saja dan harus dapat ditangani secara otomatis tanpa menghentikan seluruh sistem.

Karakteristik Utama Cloud-Native Application

Cloud-native application memiliki beberapa karakteristik kunci, antara lain:

1. **Terdistribusi**. Aplikasi terdiri dari banyak komponen kecil yang berjalan secara terpisah.

2. **Elastis**. Sistem dapat menambah atau mengurangi sumber daya sesuai kebutuhan.
3. **Tahan terhadap kegagalan**. Gangguan pada satu komponen tidak menghentikan keseluruhan aplikasi.
4. **Otomatis**. Proses *deployment*, *scaling*, dan pemulihan berjalan secara otomatis.

Karakteristik ini membuat cloud-native application lebih siap menghadapi beban kerja yang berubah-ubah.

Peran Microservices dalam Cloud-Native

Sebagian besar cloud-native application dibangun menggunakan arsitektur microservices. Setiap layanan memiliki fungsi spesifik dan dapat dikembangkan serta di-*deploy* secara independen. Pendekatan ini memungkinkan tim bekerja secara paralel dan mempercepat siklus pengembangan perangkat lunak.

Jika satu layanan mengalami masalah, layanan lain tetap dapat beroperasi. Hal ini meningkatkan ketahanan sistem secara keseluruhan.

Container sebagai Fondasi Cloud-Native

Cloud-native application hampir selalu menggunakan **container** sebagai unit eksekusi. Container memungkinkan aplikasi dan seluruh dependensinya dikemas menjadi satu kesatuan yang dapat dijalankan secara konsisten di berbagai lingkungan.

Dengan container, perbedaan antara lingkungan pengembangan dan produksi dapat diminimalkan, sehingga mengurangi risiko kesalahan saat aplikasi dipindahkan ke cloud

Otomatisasi dan Stateless Design

Cloud-native application dirancang untuk mendukung otomatisasi penuh. Ketika terjadi lonjakan pengguna, sistem dapat menyesuaikan kapasitas secara otomatis. Ketika terjadi kegagalan, layanan dapat diganti tanpa intervensi manual.

Selain itu, layanan cloud-native umumnya bersifat stateless, yaitu tidak menyimpan data secara lokal. Status dan data disimpan pada sistem terpisah, sehingga layanan dapat dihentikan atau diganti tanpa kehilangan informasi

Implikasi Cloud-Native terhadap Rekayasa Perangkat Lunak

Cloud-native bukan hanya pendekatan teknis, tetapi juga perubahan paradigma dalam rekayasa perangkat lunak. Pengembang tidak lagi berfokus pada stabilitas satu server, melainkan pada ketahanan sistem secara keseluruhan.

Pemahaman terhadap konsep cloud-native application menjadi landasan penting untuk membahas teknologi pendukung seperti container, orkestrasi, serta praktik CI/CD yang akan dibahas pada subbab berikutnya.

C. Container dan Virtualisasi

Virtualisasi merupakan teknologi penting yang memungkinkan satu perangkat keras fisik menjalankan beberapa sistem operasi melalui virtual machine (VM). Dengan bantuan *hypervisor*, sumber

daya fisik dapat dibagi secara efisien, sehingga meningkatkan pemanfaatan infrastruktur dan menjadi fondasi awal komputasi awan (Mell & Grance, 2011).

Namun, mesin virtual memiliki keterbatasan karena setiap VM menjalankan sistem operasi lengkap. Akibatnya, konsumsi sumber daya relatif tinggi dan waktu *startup* lebih lambat, sehingga kurang efisien untuk lingkungan pengembangan modern yang menuntut skalabilitas cepat dan deployment berulang. Keterbatasan ini mendorong munculnya containerization sebagai pendekatan yang lebih ringan (Pahl, 2015).

Container merupakan virtualisasi tingkat sistem operasi yang memungkinkan aplikasi dijalankan dalam lingkungan terisolasi tanpa membawa sistem operasi lengkap. Container berbagi kernel yang sama, tetapi tetap terisolasi dari sisi proses, jaringan, dan sistem berkas. Dengan mengemas aplikasi beserta dependensinya dalam satu unit portabel, container memungkinkan aplikasi berjalan konsisten di berbagai lingkungan, dari lokal hingga cloud (Merkel, 2014).

Perbedaan utama antara VM dan container terletak pada efisiensi:

- Virtual Machine lebih berat karena menyertakan sistem operasi lengkap.
- Container lebih ringan, *startup* cepat, dan lebih efisien untuk sistem cloud-native.

Dalam arsitektur cloud-native, container menjadi fondasi utama karena mendukung *immutable infrastructure* dan penerapan microservices. Setiap layanan dijalankan dalam container terpisah dan dapat dikelola secara independen, meningkatkan konsistensi dan

fleksibilitas sistem. Pemahaman tentang container dan virtualisasi ini menjadi dasar penting untuk memahami orkestrasi container, yang mengelola skala besar container secara terkoordinasi pada tahap selanjutnya.

D. Orkestrasi Container

Seiring meningkatnya penggunaan container dalam arsitektur cloud-native, jumlah container yang dijalankan dalam satu sistem dapat mencapai puluhan, ratusan, bahkan ribuan. Mengelola container secara manual dalam skala seperti ini menjadi tidak realistis. Tantangan seperti penjadwalan container, pemantauan kesehatan layanan, distribusi beban, dan pemulihan ketika terjadi kegagalan membutuhkan mekanisme otomatis dan terkoordinasi. Kebutuhan inilah yang melahirkan konsep orkestrasi container (Burns, Grant, Oppenheimer, Brewer, & Wilkes, 2016).

Orkestrasi container adalah proses pengelolaan otomatis siklus hidup container, mulai dari penempatan (*deployment*), penskalaan, pemantauan, hingga pemulihan ketika terjadi kegagalan. Sistem orkestrasi bertindak sebagai pengendali pusat yang memastikan container berjalan sesuai dengan kondisi yang diinginkan, meskipun lingkungan infrastruktur bersifat dinamis dan berubah-ubah.

Dalam konteks rekayasa perangkat lunak, orkestrasi container memungkinkan pengembang dan operator sistem fokus pada perilaku aplikasi, bukan pada detail teknis pengelolaan container satu per satu (Pahl, 2015).

Fungsi Utama Orkestrasi Container

Sistem orkestrasi container umumnya menyediakan beberapa fungsi utama, antara lain:

1. **Deployment otomatis**. Container dapat dijalankan dan diperbarui tanpa menghentikan sistem secara keseluruhan.
2. **Load balancing**. Permintaan pengguna didistribusikan ke container yang tersedia secara merata.
3. **Auto-scaling**. Jumlah container dapat bertambah atau berkurang sesuai beban kerja.
4. **Self-healing**. Container yang gagal akan diganti secara otomatis.
5. **Manajemen konfigurasi dan jaringan.** Container dapat saling berkomunikasi secara aman dan terstruktur.

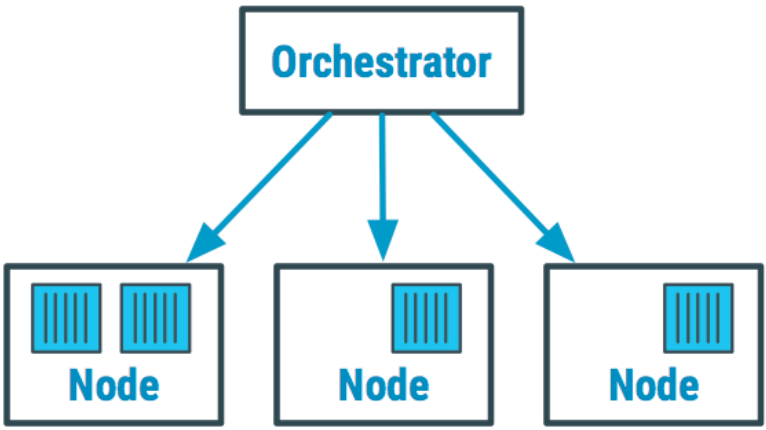
Fungsi-fungsi ini menjadikan orkestrasi container sebagai elemen kunci dalam sistem berskala besar.

Orkestrasi Container dalam Arsitektur Cloud-Native

Dalam arsitektur cloud-native, orkestrasi container berfungsi sebagai penghubung antara aplikasi dan infrastruktur cloud. Aplikasi tidak perlu mengetahui detail server atau mesin virtual, karena sistem orkestrasi memastikan container dijalankan pada lingkungan yang tersedia dan optimal. Pendekatan ini mendukung declarative configuration, di mana pengembang mendefinisikan kondisi yang diinginkan, sementara sistem orkestrasi menangani realisasinya secara otomatis (Burns et al., 2016).

Orkestrasi container berdampak langsung pada keandalan dan skalabilitas sistem. Dari sisi keandalan, sistem mampu melakukan

pemulihan otomatis dengan mengganti container yang gagal tanpa intervensi manual, sehingga mengurangi waktu henti layanan. Dari sisi skalabilitas, orkestrasi memungkinkan penyesuaian kapasitas secara dinamis sesuai beban kerja, baik dengan menambah maupun mengurangi jumlah container. Kemampuan ini menjadikan orkestrasi container komponen kunci dalam mendukung aplikasi modern yang bersifat elastis dan memiliki pola penggunaan yang berubah-ubah.



Gambar 10. Arsitektur Orkestrasi Container dengan Orchestrator dan Node

Gambar tersebut menggambarkan hubungan antara *orchestrator* dan beberapa *node* dalam sistem orkestrasi container. *Orchestrator* berfungsi sebagai pusat pengendali yang mengatur bagaimana container dijalankan, dipantau, dan dipulihkan di seluruh node yang tersedia dalam suatu klaster.

Makna Setiap Komponen pada Gambar

1. **Orchestrator:** Orchestrator merupakan komponen pusat yang bertanggung jawab menentukan *di mana* dan *bagaimana*

container dijalankan. Ia mengelola penjadwalan container, pemantauan kesehatan layanan, serta pengambilan keputusan ketika terjadi kegagalan pada node atau container tertentu.

2. **Node:** Node adalah mesin fisik atau virtual yang menyediakan sumber daya komputasi untuk menjalankan container. Setiap node dapat menjalankan satu atau lebih container sesuai dengan kapasitas sumber daya yang tersedia.

3. **Arah Panah dari Orchestrator ke Node:** Panah menunjukkan bahwa keputusan operasional berasal dari orchestrator, bukan dari node. Orchestrator mendistribusikan container ke node, memindahkan beban kerja jika diperlukan, dan memastikan bahwa kondisi sistem tetap sesuai dengan konfigurasi yang diinginkan.

Interpretasi Arsitektural

Diagram ini menegaskan prinsip utama orkestrasi container, yaitu kontrol terpusat dengan eksekusi terdistribusi. Orchestrator tidak menjalankan aplikasi secara langsung, tetapi mengatur bagaimana node menjalankan container. Dengan pendekatan ini, sistem dapat tetap berjalan meskipun salah satu node mengalami kegagalan, karena orchestrator dapat menjadwalkan ulang container ke node lain yang masih tersedia.

Dalam praktik cloud-native, pola ini memungkinkan:

- penskalaan otomatis layanan,
- pemulihan mandiri (*self-healing*),
- dan pemanfaatan sumber daya yang lebih efisien.

Dengan adanya orkestrasi container, rekayasa perangkat lunak bergeser dari pengelolaan infrastruktur manual menuju pengelolaan sistem berbasis kebijakan dan otomatisasi. Tim pengembang dan operasional dapat bekerja lebih efisien dengan mengandalkan sistem orkestrasi untuk menjaga stabilitas dan kinerja aplikasi.

Pemahaman terhadap orkestrasi container menjadi landasan penting sebelum membahas praktik continuous integration dan continuous deployment (CI/CD), yang memanfaatkan orkestrasi untuk mendukung siklus pengembangan dan rilis perangkat lunak secara berkelanjutan pada subbab berikutnya.

E. Continuous Integration dan Continuous Deployment

Perkembangan aplikasi modern yang cepat dan kompleks menuntut proses pengembangan yang efisien dan andal. Pendekatan tradisional dengan integrasi manual dan rilis berkala sering menimbulkan konflik kode serta keterlambatan rilis. Untuk mengatasinya, praktik Continuous Integration (CI) dan Continuous Deployment (CD) dikembangkan sebagai bagian dari budaya DevOps yang menekankan otomatisasi dan kolaborasi (Fowler & Humble, 2010).

Continuous Integration (CI) adalah praktik penggabungan perubahan kode secara rutin ke repositori bersama, yang secara otomatis diuji melalui kompilasi, pengujian unit, dan pemeriksaan kualitas kode. Tujuan utamanya adalah mendeteksi kesalahan sejak dini, mengurangi konflik integrasi, dan meningkatkan stabilitas kode. Dengan CI, setiap perubahan diverifikasi sebelum digabungkan ke cabang utama, sehingga kualitas sistem dapat dijaga secara konsisten (Humble & Farley, 2010).

Continuous Deployment (CD) merupakan kelanjutan dari CI, di mana perubahan kode yang lolos seluruh pengujian otomatis langsung diterapkan ke lingkungan produksi. Pendekatan ini memungkinkan rilis fitur dan perbaikan dilakukan secara cepat, bertahap, dan berisiko lebih rendah. Namun, CD menuntut pengujian dan monitoring yang sangat andal agar kesalahan tidak berdampak langsung pada pengguna.

Dalam arsitektur cloud-native, CI/CD berperan sebagai penghubung antara pengembangan dan operasional. Pipeline CI/CD terintegrasi dengan container dan orkestrasi, memungkinkan proses build, test, dan deploy berjalan otomatis. Praktik seperti *rolling update* dan *blue-green deployment* membantu menjaga ketersediaan layanan selama rilis sistem (Bass et al., 2022).

F. Arsitektur Multi-Cloud dan Hybrid Cloud

Meningkatnya ketergantungan organisasi pada komputasi awan mendorong kebutuhan untuk mengurangi *vendor lock-in*, meningkatkan ketahanan sistem, serta memenuhi tuntutan regulasi dan keamanan. Kondisi ini melahirkan adopsi arsitektur multi-cloud dan hybrid cloud, yang memungkinkan pemanfaatan beberapa lingkungan cloud secara terkoordinasi (Bass, Clements, & Kazman, 2022).

Multi-cloud merupakan pendekatan penggunaan lebih dari satu penyedia cloud publik dalam satu sistem. Setiap penyedia dipilih berdasarkan keunggulan tertentu, seperti performa, layanan spesifik, atau lokasi geografis. Pendekatan ini memberikan fleksibilitas tinggi dan mengurangi ketergantungan pada satu vendor, tetapi menuntut perencanaan arsitektural yang matang agar integrasi dan konsistensi antar lingkungan tetap terjaga (Richards & Ford, 2020).

Sementara itu, hybrid cloud mengombinasikan cloud publik dengan private cloud atau infrastruktur lokal (*on-premises*). Model ini memungkinkan data sensitif dan sistem kritis tetap berada di lingkungan privat, sementara beban kerja lain dijalankan di cloud publik untuk efisiensi dan skalabilitas. Hybrid cloud banyak diterapkan pada sektor dengan regulasi ketat, seperti keuangan, kesehatan, dan pemerintahan (Mell & Grance, 2011).

Perbedaan utama kedua pendekatan tersebut dapat diringkas sebagai berikut:

- Multi-cloud berfokus pada fleksibilitas dan pengurangan ketergantungan vendor melalui penggunaan beberapa cloud publik.
- Hybrid cloud menekankan keseimbangan antara keamanan, regulasi, dan fleksibilitas dengan menggabungkan cloud publik dan privat.

BAB V
SKALABILITAS SISTEM PERANGKAT LUNAK

Pertumbuhan jumlah pengguna, lonjakan volume data, dan tuntutan ketersediaan layanan 24/7 menjadikan skalabilitas sebagai isu krusial dalam rekayasa perangkat lunak modern, karena banyak sistem yang tampak sukses pada tahap awal justru runtuh ketika beban meningkat akibat arsitektur yang tidak dirancang untuk berkembang. Karena itu, skalabilitas kini diperlakukan sebagai kebutuhan fundamental yang harus dipertimbangkan sejak perancangan awal, bukan sekadar fitur tambahan. Bab ini mengulas konsep dan strategi skalabilitas secara menyeluruh, mencakup penskalaan horizontal dan vertikal, load balancing dan auto-scaling, serta pengelolaan data dan identifikasi bottleneck, sekaligus membahas prinsip high availability dan fault tolerance sebagai dasar sistem yang tangguh terhadap kegagalan, sehingga pembaca memahami cara merancang dan mengelola sistem yang dapat tumbuh berkelanjutan tanpa mengorbankan kinerja dan keandalan.

A. Konsep Skalabilitas Horizontal dan Vertikal

Skalabilitas adalah kemampuan suatu sistem perangkat lunak untuk menangani peningkatan beban kerja baik berupa jumlah pengguna, permintaan layanan, maupun volume datatanpa mengalami penurunan kinerja yang signifikan. Sistem yang skalabel dapat berkembang seiring pertumbuhan kebutuhan bisnis dan teknologi tanpa harus dilakukan perombakan besar pada arsitekturnya.

Dalam rekayasa perangkat lunak modern, skalabilitas tidak hanya berkaitan dengan kapasitas teknis, tetapi juga dengan efisiensi biaya, keandalan layanan, dan keberlanjutan sistem dalam jangka panjang.

Skalabilitas Vertikal (Vertical Scaling)

Skalabilitas vertikal, sering disebut *scale up*, adalah pendekatan meningkatkan kapasitas sistem dengan menambah sumber daya pada satu mesin. Penambahan ini dapat berupa peningkatan CPU, memori, penyimpanan, atau kemampuan jaringan pada server yang sama.

Pendekatan ini relatif sederhana untuk diterapkan karena tidak memerlukan perubahan signifikan pada arsitektur aplikasi. Namun, skalabilitas vertikal memiliki keterbatasan fisik dan ekonomis. Kapasitas perangkat keras tidak dapat ditingkatkan tanpa batas, dan peningkatan sumber daya pada satu mesin sering kali berbiaya tinggi serta berisiko menimbulkan *single point of failure*.

Skalabilitas Horizontal (Horizontal Scaling)

Skalabilitas horizontal, atau *scale out*, adalah pendekatan meningkatkan kapasitas sistem dengan menambahkan lebih banyak mesin atau instans layanan. Beban kerja kemudian didistribusikan ke banyak node melalui mekanisme seperti load balancer.

Pendekatan ini sangat sesuai dengan arsitektur terdistribusi dan cloud-native karena memungkinkan sistem tumbuh secara elastis. Ketika beban meningkat, instans baru dapat ditambahkan; ketika beban menurun, instans dapat dikurangi. Skalabilitas horizontal juga meningkatkan ketahanan sistem, karena kegagalan satu node tidak serta-merta menghentikan keseluruhan layanan.

Perbandingan Horizontal dan Vertikal

Perbedaan mendasar antara kedua pendekatan ini dapat dirangkum sebagai berikut:

- **Skalabilitas Vertikal**
    - Lebih sederhana secara arsitektural
    - Cepat diterapkan pada sistem kecil
    - Terbatas oleh kapasitas perangkat keras
    - Berisiko menjadi titik kegagalan tunggal
- **Skalabilitas Horizontal**
    - Lebih kompleks secara arsitektural
    - Sangat fleksibel dan elastis
    - Mendukung sistem skala besar
    - Lebih tahan terhadap kegagalan

Dalam praktik modern, banyak sistem mengombinasikan kedua pendekatan ini untuk memperoleh keseimbangan antara kemudahan implementasi dan kemampuan tumbuh jangka panjang.

Skalabilitas dalam Arsitektur Modern

Arsitektur cloud-native dan microservices secara alami lebih mendukung skalabilitas horizontal. Setiap layanan dapat diskalakan secara independen sesuai kebutuhan. Pendekatan ini memungkinkan sistem menangani lonjakan beban secara dinamis tanpa harus meningkatkan kapasitas seluruh sistem.

Sebaliknya, sistem monolitik tradisional cenderung mengandalkan skalabilitas vertikal, yang cepat mencapai batas kapasitas ketika beban meningkat tajam. Pemilihan pendekatan skalabilitas harus dilakukan sejak tahap perancangan arsitektur. Sistem yang dirancang tanpa mempertimbangkan skalabilitas sejak awal akan sulit dikembangkan ketika kebutuhan meningkat.

B. Load Balancing dan Auto-Scaling

Ketika sebuah sistem perangkat lunak diskalakan secara horizontal, permintaan pengguna tidak lagi ditangani oleh satu server saja, melainkan oleh banyak instans layanan. Dalam kondisi ini, diperlukan mekanisme untuk mendistribusikan beban kerja secara merata, agar tidak terjadi penumpukan permintaan pada satu instans tertentu. Mekanisme tersebut dikenal sebagai load balancing.

Load balancing berfungsi sebagai pengatur lalu lintas permintaan yang masuk, memastikan setiap instans layanan menerima beban sesuai kapasitasnya. Tanpa load balancing, penambahan instans tidak akan efektif karena sebagian instans dapat kelebihan beban sementara instans lain tidak dimanfaatkan secara optimal.

Prinsip Kerja Load Balancer

Load balancer berada di antara klien dan layanan backend. Setiap permintaan yang datang akan diteruskan ke salah satu instans layanan berdasarkan algoritma tertentu. Beberapa pendekatan umum yang digunakan antara lain:

- **Round-robin**, yaitu permintaan dibagikan secara bergiliran ke setiap instans.

- **Least connections**, yaitu permintaan diarahkan ke instans dengan jumlah koneksi aktif paling sedikit.
- **Weighted distribution**, yaitu pembagian beban berdasarkan kapasitas masing-masing instans.

Dengan pendekatan ini, load balancer membantu menjaga stabilitas kinerja dan mencegah satu instans menjadi titik kemacetan sistem.

Konsep Auto-Scaling

Auto-scaling adalah mekanisme yang memungkinkan sistem menyesuaikan jumlah instans layanan secara otomatis berdasarkan kondisi beban kerja. Ketika permintaan meningkat, sistem dapat menambah instans baru; ketika permintaan menurun, instans yang tidak diperlukan dapat dihentikan untuk menghemat sumber daya.

Auto-scaling sangat penting dalam lingkungan cloud karena beban kerja sering kali bersifat fluktuatif. Tanpa auto-scaling, sistem harus disiapkan untuk beban maksimum sepanjang waktu, yang berakibat pada pemborosan sumber daya dan biaya operasional yang tinggi

Hubungan Load Balancing dan Auto-Scaling

Load balancing dan auto-scaling bekerja secara saling melengkapi. Auto-scaling menentukan berapa banyak instans yang harus tersedia, sedangkan load balancing menentukan bagaimana permintaan dibagikan ke instans-instans tersebut. Tanpa load balancing, instans tambahan hasil auto-scaling tidak akan dimanfaatkan secara efektif. Sebaliknya, tanpa auto-scaling, load balancing hanya membagi beban pada kapasitas yang tetap.

Kombinasi keduanya memungkinkan sistem menangani lonjakan trafik secara dinamis tanpa gangguan layanan.

Load Balancing dan Auto-Scaling dalam Arsitektur Modern

Dalam arsitektur cloud-native dan microservices, load balancing dan auto-scaling sering diintegrasikan dengan orkestrasi container. Setiap layanan dapat diskalakan secara independen berdasarkan metrik tertentu seperti penggunaan CPU, memori, atau jumlah permintaan per detik.

Pendekatan ini memungkinkan sistem beradaptasi secara real-time terhadap perubahan beban dan meningkatkan ketahanan layanan, karena kegagalan satu instans dapat segera ditangani dengan penggantian instans baru.

Implikasi terhadap Kinerja dan Biaya

Penerapan load balancing dan auto-scaling yang tepat tidak hanya meningkatkan kinerja sistem, tetapi juga membantu mengendalikan biaya operasional. Sistem hanya menggunakan sumber daya sesuai kebutuhan aktual, bukan berdasarkan perkiraan beban maksimum. Namun, konfigurasi yang kurang tepat dapat menimbulkan masalah seperti respons lambat atau pembengkakan biaya akibat penskalaan berlebihan. Oleh karena itu, perancangan load balancing dan auto-scaling harus mempertimbangkan karakteristik beban kerja dan tujuan sistem secara menyeluruh.

C. Caching dan Content Delivery Network

Dalam sistem perangkat lunak berskala besar, banyak permintaan pengguna bersifat berulang, misalnya permintaan halaman utama, data profil, atau konten statis. Jika setiap permintaan selalu diproses

langsung oleh server aplikasi atau basis data, beban sistem akan meningkat secara signifikan dan berdampak pada penurunan kinerja. Untuk mengatasi hal ini, digunakan mekanisme caching, yaitu penyimpanan sementara data yang sering diakses agar dapat disajikan lebih cepat.

Caching bertujuan untuk mengurangi beban komputasi, mempercepat waktu respons, dan meningkatkan pengalaman pengguna. Dalam konteks skalabilitas, caching berperan sebagai lapisan pelindung yang mencegah sistem inti terbebani oleh permintaan yang sebenarnya dapat dilayani dari data yang sudah tersedia.

Caching bekerja dengan menyimpan salinan data hasil pemrosesan sebelumnya di lokasi yang lebih dekat atau lebih cepat diakses. Ketika permintaan yang sama muncul kembali, sistem akan memeriksa cache terlebih dahulu sebelum memproses permintaan tersebut ke sumber utama.

Secara umum, cache dapat diterapkan pada berbagai lapisan sistem, mulai dari:

- cache di sisi klien (browser),
- cache di sisi aplikasi,
- cache di lapisan basis data,
- hingga cache terdistribusi yang digunakan bersama oleh banyak layanan.

Dengan pendekatan ini, sistem dapat mengurangi latensi dan meningkatkan throughput secara signifikan.

Jenis-Jenis Caching

Dalam praktik rekayasa perangkat lunak, caching dapat diklasifikasikan menjadi beberapa jenis utama:

1. **Cache statis**. Digunakan untuk konten yang jarang berubah, seperti gambar, file CSS, atau JavaScript.
2. **Cache dinamis**. Digunakan untuk data hasil pemrosesan aplikasi yang sering diminta ulang.
3. **Cache terdistribusi.** Digunakan pada sistem berskala besar agar cache dapat diakses oleh banyak instans layanan secara konsisten.

Pemilihan jenis cache harus mempertimbangkan karakteristik data dan tingkat perubahan informasi

Content Delivery Network (CDN)

Content Delivery Network (CDN) adalah jaringan server terdistribusi secara geografis yang digunakan untuk menyajikan konten kepada pengguna dari lokasi terdekat. CDN menyimpan salinan konten statis dan semi-statis di berbagai titik (*edge locations*) sehingga permintaan pengguna tidak perlu selalu diarahkan ke server pusat.

Dengan CDN, waktu respons menjadi lebih cepat dan beban server utama berkurang. CDN sangat efektif untuk aplikasi dengan pengguna global, seperti platform media, e-commerce, dan layanan pendidikan daring.

Hubungan Caching dan CDN

Caching dan CDN saling melengkapi dalam mendukung skalabilitas sistem. Caching berfokus pada pengurangan beban pemrosesan di dalam sistem, sementara CDN berfokus pada pengurangan latensi jaringan dan distribusi konten secara global. Kombinasi keduanya memungkinkan sistem menangani jumlah pengguna yang besar tanpa peningkatan infrastruktur yang signifikan.

Dalam arsitektur modern, CDN sering dipandang sebagai bentuk caching tingkat lanjut yang berada di lapisan terluar sistem.

Meskipun caching memberikan banyak manfaat, penerapannya juga menghadirkan tantangan, terutama terkait konsistensi data. Data yang disimpan dalam cache dapat menjadi usang jika sumber data utama berubah. Oleh karena itu, diperlukan strategi pengelolaan cache seperti pengaturan masa berlaku (*time-to-live*), pembaruan otomatis, atau penghapusan cache secara selektif.

Kesalahan dalam pengelolaan cache dapat menyebabkan pengguna menerima data yang tidak akurat, sehingga perancangan mekanisme caching harus dilakukan dengan cermat.

D. Database Scalability

Dalam sistem perangkat lunak skala besar, basis data sering menjadi bottleneck utama karena menyimpan *state* sistem dan bertanggung jawab atas konsistensi data. Sementara lapisan aplikasi dapat diskalakan secara horizontal dengan relatif mudah, basis data memerlukan strategi khusus agar mampu menangani peningkatan beban tanpa menurunkan kinerja dan keandalan layanan (Stonebraker et al., 2010).

Pendekatan paling sederhana adalah skalabilitas vertikal, yaitu meningkatkan kapasitas satu server basis data dengan menambah CPU, memori, atau penyimpanan. Cara ini mudah diterapkan, tetapi memiliki batas fisik, biaya tinggi, dan berisiko menciptakan *single point of failure* jika tidak disertai redundansi (Bass, Clements, & Kazman, 2022).

Untuk meningkatkan kapasitas dan ketersediaan, banyak sistem menggunakan replikasi basis data, di mana data disalin ke beberapa replika. Replikasi efektif untuk mendistribusikan beban baca, tetapi sering kali bersifat asinkron sehingga menimbulkan tantangan konsistensi dan mengharuskan penerimaan *eventual consistency* dalam kondisi tertentu (Vogels, 2009).

Pendekatan lain adalah sharding, yaitu pembagian data ke beberapa basis data berdasarkan kunci partisi tertentu. Sharding memungkinkan skalabilitas horizontal yang lebih baik untuk beban baca dan tulis, tetapi meningkatkan kompleksitas aplikasi, terutama untuk operasi lintas shard dan pemilihan kunci partisi yang tepat (Stonebraker et al., 2010).

Perkembangan sistem terdistribusi mendorong penggunaan basis data NoSQL dan terdistribusi, yang dirancang untuk skalabilitas horizontal dan toleransi kegagalan. Sistem ini sering mengorbankan sebagian konsistensi demi ketersediaan, sesuai dengan prinsip CAP theorem, dan cocok untuk aplikasi berskala besar seperti media sosial dan analitik (Brewer, 2012).

Contoh Kasus Skalabilitas Database

1. Skalabilitas Vertikal (Vertical Scaling)

Kasus: Sebuah aplikasi akademik kampus awalnya memiliki 1 database server MySQL yang menyimpan data mahasiswa, nilai, dan KRS. Seiring bertambahnya jumlah mahasiswa, query menjadi lambat terutama saat pengisian KRS.

Solusi yang diambil: Tim IT meningkatkan spesifikasi server:

- RAM dari 16 GB → 64 GB
- CPU dari 8 core → 32 core
- Storage dari HDD → NVMe SSD

Karakteristik utama

- Tetap satu database, satu server
- Tidak ada perubahan desain aplikasi
- Skalabilitas terbatas oleh hardware
- Jika server mati → seluruh sistem berhenti

Skalabilitas vertikal = *"memperbesar mesin yang sama."*

Gambar 11. Skalabilitas vertikal

2. Skalabilitas Sharding (Horizontal Scaling pada Data)

Kasus: Platform e-commerce nasional mulai kewalahan karena tabel orders berisi ratusan juta transaksi. Query berdasarkan user_id semakin lambat meskipun server sudah sangat besar.

Solusi yang diambil: Database di-*shard* berdasarkan wilayah pengguna:

- Shard 1 → pengguna Sumatera
- Shard 2 → pengguna Jawa
- Shard 3 → pengguna Kalimantan dan Indonesia Timur

Setiap shard:

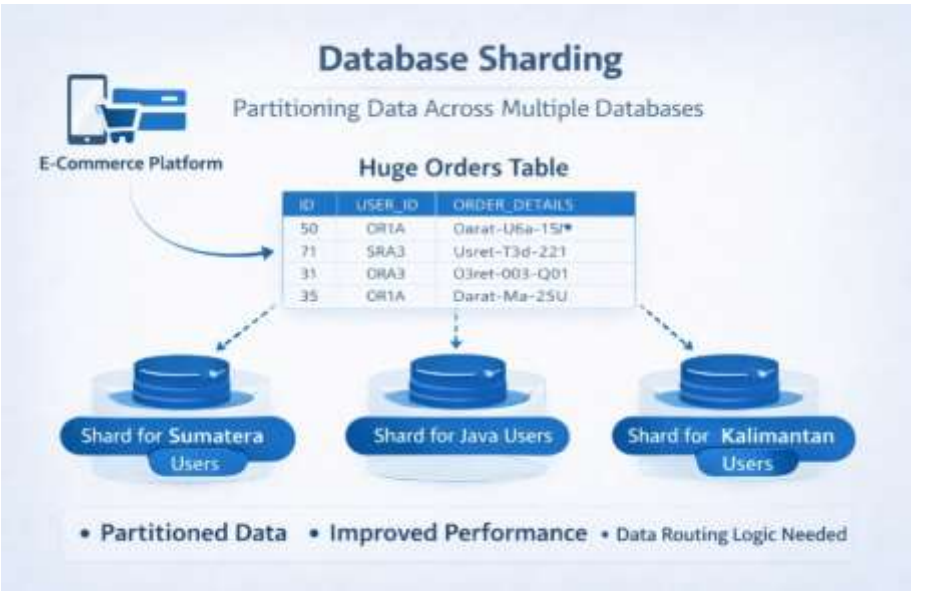
- Memiliki database terpisah

- Menyimpan subset data berbeda
- Query diarahkan ke shard yang relevan

Karakteristik utama

- Data dibagi (partitioned), bukan digandakan
- Beban database tersebar
- Aplikasi harus tahu aturan routing shard
- Kompleks saat:
    - pindah shard
    - agregasi lintas shard

Sharding = *"membagi data agar beban terbagi"*.



Gambar 12. Skalabilitas horizontal (sharding)

3. Sistem Terdistribusi (Distributed Database System)

Kasus: Aplikasi ride-hailing dan pembayaran digital harus:

- Tetap aktif 24/7
- Melayani jutaan transaksi real-time
- Tahan terhadap kegagalan data center

Solusi yang diambil: Menggunakan database terdistribusi:

- Data direplikasi ke beberapa node
- Node tersebar di beberapa lokasi geografis
- Sistem tetap berjalan meski sebagian node gagal

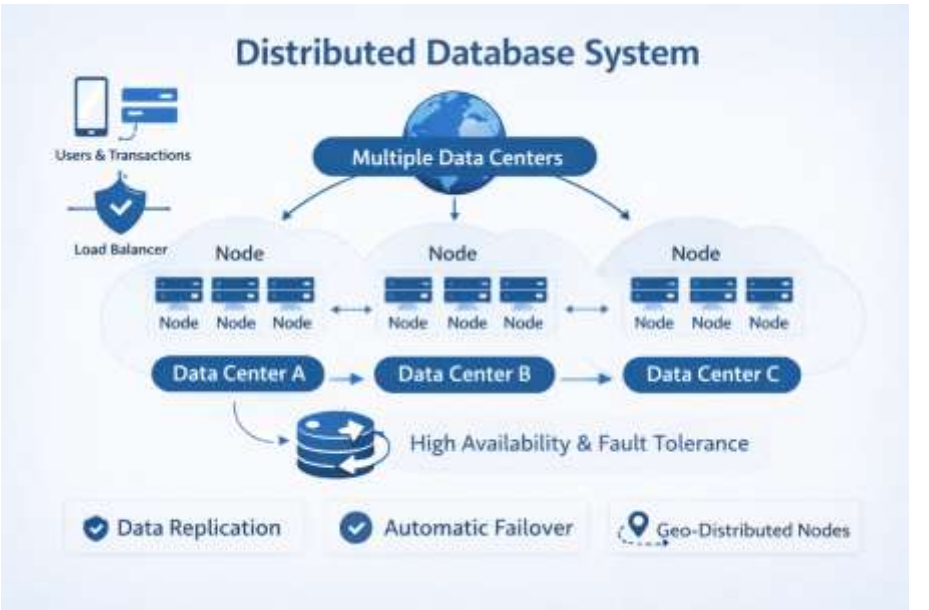
Contoh implementasi:

- Data transaksi disimpan di beberapa node sekaligus
- Konsistensi dijaga dengan protokol tertentu
- Sistem otomatis memilih node yang sehat

Karakteristik utama

- Banyak node aktif secara bersamaan
- Mendukung high availability
- Lebih kompleks (konsistensi, latensi, sinkronisasi)
- Cocok untuk sistem kritikal dan skala besar

Sistem terdistribusi = *"banyak database bekerja sebagai satu sistem"*.



Gambar 13. Database Terdistribusi

Ringkasan Perbandingan Skalabilitas Database

Aspek	**Skalabilitas Vertikal**	**Database Sharding**	**Sistem Terdistribusi**
Jumlah Server	1 server	Banyak server	Banyak server
Pengelolaan Data	Utuh di satu server	Data dibagi (partition)	Data dibagi + direplikasi
Fokus Utama	Performa hardware	Distribusi beban data	Ketersediaan & ketahanan
Kompleksitas Teknis	Rendah	Sedang	Tinggi
Single Point of Failure	Tinggi	Sedang	Rendah
Skalabilitas Jangka Panjang	Terbatas	Baik	Sangat baik
Biaya	Rendah → tinggi	Sedang	Tinggi

Aspek	Skalabilitas Vertikal	Database Sharding	Sistem Terdistribusi
Infrastruktur	(hardware mahal)		
Biaya Operasional	Rendah	Sedang	Tinggi
Biaya SDM & Keahlian	Rendah	Sedang	Tinggi
Efisiensi Biaya pada Skala Besar	Rendah	Baik	Sangat baik (jangka panjang)
Contoh Penggunaan	Aplikasi kecil	E-commerce	Fintech, cloud, critical system

Skalabilitas vertikal memperbesar satu database, sharding membagi data ke beberapa database, sedangkan sistem terdistribusi membangun ekosistem database yang saling bekerja sama untuk ketahanan dan ketersediaan tinggi.

E. Bottleneck Analysis

Bottleneck adalah titik dalam sistem perangkat lunak yang membatasi kinerja keseluruhan, meskipun komponen lain masih memiliki kapasitas yang tersedia. Dalam sistem berskala besar, peningkatan sumber daya pada satu bagian tidak selalu meningkatkan kinerja total jika bagian lain tetap menjadi penghambat. Oleh karena itu, memahami dan mengidentifikasi bottleneck merupakan langkah krusial dalam rekayasa kinerja dan skalabilitas sistem.

Jenis-Jenis Bottleneck

Bottleneck dapat muncul pada berbagai lapisan sistem, antara lain:

1. **Bottleneck komputasi**. Terjadi ketika CPU atau memori menjadi faktor pembatas akibat proses yang intensif.

2. **Bottleneck jaringan**. Muncul akibat latensi tinggi, bandwidth terbatas, atau konfigurasi jaringan yang tidak optimal.

3. **Bottleneck basis data**. Terjadi ketika operasi baca/tulis melebihi kapasitas basis data, misalnya akibat kueri yang tidak efisien.

4. **Bottleneck I/O**. Berkaitan dengan akses penyimpanan yang lambat, seperti disk atau sistem file terdistribusi.

5. **Bottleneck aplikasi**. Disebabkan oleh desain kode yang tidak efisien, seperti penggunaan *synchronous calls* berlebihan.

Identifikasi jenis bottleneck membantu menentukan solusi yang tepat dan terarah.

Prinsip Dasar Bottleneck Analysis

Analisis bottleneck berangkat dari prinsip bahwa kinerja sistem ditentukan oleh komponen paling lambat. Prinsip ini sejalan dengan hukum dasar teori antrean, di mana peningkatan beban pada satu komponen yang sudah jenuh akan menyebabkan antrean dan latensi meningkat secara tidak proporsional.

Oleh karena itu, optimasi sistem harus difokuskan pada titik bottleneck, bukan pada bagian sistem yang masih memiliki kapasitas berlebih.

Teknik Identifikasi Bottleneck

Beberapa teknik umum yang digunakan dalam bottleneck analysis meliputi:

- **Monitoring dan metrik kinerja**. Mengamati penggunaan CPU, memori, latensi, dan throughput secara real-time.
- **Profiling aplikasi**. Mengidentifikasi fungsi atau proses yang paling banyak mengonsumsi sumber daya.
- **Load testing dan stress testing**. Mensimulasikan beban tinggi untuk melihat bagaimana sistem bereaksi.
- **Tracing terdistribusi**. Melacak alur permintaan lintas layanan untuk menemukan titik keterlambatan.

Pendekatan ini memungkinkan analisis berbasis data, bukan asumsi semata.

Bottleneck Analysis dalam Sistem Terdistribusi

Pada sistem terdistribusi dan microservices, bottleneck sering kali tidak terlihat secara langsung. Keterlambatan kecil pada satu layanan dapat berdampak besar pada keseluruhan alur proses. Oleh karena itu, analisis bottleneck harus dilakukan secara menyeluruh dengan mempertimbangkan interaksi antar layanan, dependensi jaringan, dan pola komunikasi sistem (Bass, Clements, & Kazman, 2022).

BAB VI
ARSITEKTUR DATA DAN SISTEM TERDISTRIBUSI

Perkembangan sistem perangkat lunak modern sangat ditentukan oleh ledakan data dan kebutuhan untuk memprosesnya secara cepat, andal, dan terdistribusi, sehingga aplikasi besar seperti e-commerce, media sosial, dan layanan keuangan kini bergantung pada arsitektur data terdistribusi alih-alih satu basis data terpusat, yang sekaligus memunculkan tantangan baru terkait konsistensi, ketersediaan, dan toleransi kegagalan. Bab ini menguraikan dasar konseptual dan praktis arsitektur data terdistribusi, mulai dari karakteristik sistem terdistribusi dan persoalan konsistensi, CAP Theorem sebagai kerangka keputusan arsitektural, strategi replikasi dan sharding, peran event streaming dan message broker untuk pemrosesan real-time, hingga arsitektur big data berskala sangat besar, serta menutupnya dengan pembahasan data governance sebagai kunci untuk menjaga kualitas, keamanan, dan kepatuhan data dalam sistem yang semakin kompleks.

A. Sistem Terdistribusi dan Konsistensi Data

Sistem terdistribusi adalah sistem perangkat lunak yang terdiri dari banyak komponen independen yang berjalan pada beberapa komputer atau node berbeda, tetapi bekerja bersama seolah-olah sebagai satu sistem tunggal. Komponen-komponen ini saling berkomunikasi melalui jaringan untuk berbagi data dan menyelesaikan tugas bersama. Contoh nyata sistem terdistribusi meliputi platform e-commerce global, media sosial, layanan keuangan digital, dan sistem cloud modern (Tanenbaum & Van Steen, 2017).

Ciri utama sistem terdistribusi adalah tidak adanya pusat kendali tunggal. Setiap node dapat beroperasi secara mandiri, tetapi tetap bergantung pada node lain untuk menyediakan layanan yang utuh kepada pengguna.

Mengapa Sistem Terdistribusi Digunakan

Sistem terdistribusi digunakan karena mampu:

- **Meningkatkan skalabilitas**, dengan menambah node saat beban meningkat.
- **Meningkatkan ketersediaan**, karena kegagalan satu node tidak langsung menghentikan sistem.
- **Mendukung distribusi geografis**, sehingga layanan dapat diakses dengan latensi rendah di berbagai wilayah.

Namun, keuntungan ini datang bersama tantangan baru, terutama dalam pengelolaan data yang konsisten di seluruh node.

Konsistensi data mengacu pada sejauh mana semua node dalam sistem terdistribusi melihat data yang sama pada waktu yang sama. Dalam sistem terpusat, konsistensi relatif mudah dijaga karena hanya ada satu sumber kebenaran. Sebaliknya, dalam sistem terdistribusi, data sering disalin atau dipartisi, sehingga perubahan pada satu node tidak selalu langsung terlihat oleh node lain (Kleppmann, 2017).

Masalah konsistensi muncul ketika sistem harus menyeimbangkan kecepatan, ketersediaan, dan keandalan.

Model-Model Konsistensi

Beberapa model konsistensi yang umum digunakan dalam sistem terdistribusi meliputi:

1. **Strong consistency**. Setiap pembacaan data selalu menampilkan nilai terbaru. Model ini memberikan kepastian tinggi, tetapi biasanya mengorbankan kinerja dan ketersediaan.

2. **Eventual consistency**. Data akan menjadi konsisten setelah periode waktu tertentu. Model ini umum digunakan pada sistem berskala besar karena lebih toleran terhadap latensi dan kegagalan jaringan.

3. **Causal consistency**. Menjamin urutan sebab–akibat antar operasi, tetapi tidak memaksa semua node selalu sinkron secara langsung.

Pemilihan model konsistensi sangat bergantung pada kebutuhan aplikasi dan toleransi terhadap ketidaksinkronan data.

Contoh Nyata Konsistensi Data

Dalam platform e-commerce berskala besar, stok barang sering direplikasi ke berbagai pusat data. Ketika satu pengguna membeli produk, pembaruan stok mungkin tidak langsung terlihat oleh pengguna lain di wilayah berbeda. Sistem biasanya menerima *eventual consistency* untuk menjaga kinerja dan ketersediaan, dengan risiko kecil terjadinya konflik yang kemudian diselesaikan oleh sistem (Vogels, 2009).

Contoh ini menunjukkan bahwa konsistensi absolut tidak selalu menjadi tujuan utama dalam sistem terdistribusi modern

Keputusan terkait konsistensi data berdampak langsung pada arsitektur sistem. Arsitek harus menentukan:

- tingkat konsistensi yang dibutuhkan,
- toleransi terhadap keterlambatan data,
- serta dampak kegagalan jaringan.

B. CAP Theorem

Dalam sistem terdistribusi, data tidak lagi berada pada satu lokasi, melainkan tersebar di banyak node yang terhubung melalui jaringan. Kondisi ini membuat kegagalan jaringan (*network partition*) menjadi sesuatu yang tidak dapat dihindari, bukan sekadar kemungkinan. Untuk memahami batasan fundamental dari sistem semacam ini, Eric Brewer memperkenalkan CAP Theorem, yang kemudian dibuktikan secara formal oleh Gilbert dan Lynch (Brewer, 2012).

CAP Theorem menjadi kerangka dasar bagi arsitek sistem untuk memahami mengapa tidak semua tujuan sistem dapat dicapai secara bersamaan.

Tiga Pilar CAP Theorem

CAP Theorem menyatakan bahwa sistem terdistribusi tidak dapat secara simultan menjamin tiga properti berikut:

1. Consistency (C). Setiap pembacaan data akan selalu mengembalikan nilai terbaru yang telah ditulis, tanpa memandang node mana yang diakses.

2. Availability (A). Setiap permintaan yang valid akan mendapatkan respons, meskipun respons tersebut mungkin tidak berisi data terbaru.

3. Partition Tolerance (P). Sistem tetap beroperasi meskipun terjadi gangguan komunikasi antar node.

Dalam praktik sistem terdistribusi modern, *partition tolerance* hampir selalu menjadi keharusan karena jaringan tidak pernah sepenuhnya andal. Makna utama CAP Theorem bukan bahwa sistem harus "memilih dua dari tiga" secara permanen, tetapi bahwa ketika terjadi partition, sistem harus memilih antara consistency atau availability. Pilihan ini bersifat situasional dan sangat bergantung pada kebutuhan aplikasi (Brewer, 2012). Dengan kata lain, CAP Theorem memaksa pengambil keputusan arsitektural untuk menetapkan **prioritas eksplisit**, bukan asumsi implisit.

Contoh Nyata Penerapan CAP

Dalam sistem perbankan digital, konsistensi biasanya menjadi prioritas utama. Jika saldo rekening ditampilkan, nilainya harus benar dan terbaru, meskipun sistem harus menolak sementara beberapa permintaan saat terjadi gangguan jaringan. Sistem semacam ini cenderung memilih CP (Consistency + Partition Tolerance).

Sebaliknya, pada aplikasi media sosial atau e-commerce, ketersediaan layanan sering lebih penting. Sistem dapat menerima bahwa jumlah *likes* atau stok barang terlihat berbeda untuk sementara waktu di lokasi berbeda. Sistem ini cenderung memilih AP (Availability + Partition Tolerance) dengan model *eventual consistency* (Vogels, 2009)

Kesalahpahaman Umum tentang CAP

Salah satu kesalahpahaman yang sering muncul adalah anggapan bahwa sistem selalu berada pada satu kategori CAP. Pada kenyataannya, banyak sistem modern bersifat adaptif, di mana tingkat konsistensi dan ketersediaan dapat disesuaikan berdasarkan konteks operasi atau jenis data (Kleppmann, 2017). Sebagai contoh, satu sistem dapat bersifat sangat konsisten untuk transaksi finansial, tetapi lebih longgar untuk data analitik atau notifikasi.

Implikasi terhadap Arsitektur Data

CAP Theorem memberikan dasar konseptual bagi keputusan arsitektur berikutnya, termasuk:

- pemilihan jenis basis data,
- strategi replikasi dan sharding,
- serta desain mekanisme pemulihan kegagalan.

Gambar 14. Visualisasi CAP Theorem dan trade-off antara Consistency, Availability, dan Partition Tolerance dalam sistem terdistribusi.

Gambar tersebut memvisualisasikan CAP Theorem melalui tiga irisan yang merepresentasikan Consistency, Availability, dan Partition Tolerance: Consistency berarti semua node melihat data yang sama pada waktu yang sama, Availability berarti setiap permintaan selalu mendapat respons meski ada kegagalan, dan Partition Tolerance menunjukkan kemampuan sistem tetap berjalan saat terjadi gangguan komunikasi antar node. Tidak ada area yang mencakup ketiganya sekaligus, menegaskan inti CAP Theorem bahwa saat terjadi partition jaringan, sistem hanya dapat menjamin dua dari tiga properti tersebut area CP memprioritaskan konsistensi dengan mengorbankan ketersediaan sementara, area AP mengutamakan ketersediaan dengan menerima konsistensi eventual, sedangkan area CA hanya mungkin terjadi tanpa partition jaringan dan jarang relevan pada sistem berskala besar. Visualisasi ini memperjelas bahwa pilihan CAP bukan teori abstrak, melainkan konsekuensi desain nyata yang harus dipertimbangkan arsitek sistem dalam merancang arsitektur data terdistribusi.

C. Replikasi dan Sharding Data

Dalam sistem terdistribusi, satu server basis data jarang mampu menangani pertumbuhan beban baca, tulis, dan ketersediaan secara bersamaan. Untuk itu, arsitektur data modern menggunakan dua strategi utama: replikasi dan sharding. Keduanya bertujuan meningkatkan skalabilitas dan keandalan, tetapi bekerja dengan mekanisme dan konsekuensi yang berbeda (Kleppmann, 2017).

Replikasi Data

Replikasi adalah proses menyalin data yang sama ke beberapa node basis data. Umumnya terdapat satu node utama (primary) yang

menerima penulisan, dan beberapa node replika (replica) yang melayani pembacaan. Dengan cara ini, beban baca dapat didistribusikan sehingga latensi menurun dan ketersediaan meningkat.

Replikasi dapat bersifat sinkron (penulisan dianggap berhasil setelah semua replika diperbarui) atau asinkron (replika diperbarui setelah penulisan pada primary). Replikasi sinkron memberikan konsistensi lebih kuat, tetapi berdampak pada latensi; sebaliknya, replikasi asinkron meningkatkan kinerja dengan menerima kemungkinan *eventual consistency* (Vogels, 2009).

Contoh nyata: platform berita global menyalin konten ke banyak replika di berbagai wilayah agar pembaca mendapatkan akses cepat tanpa membebani satu server pusat.

Sharding Data

Berbeda dengan replikasi, sharding membagi data menjadi beberapa bagian (*shard*) berdasarkan kunci tertentu, seperti ID pengguna atau wilayah. Setiap shard disimpan dan dikelola oleh node yang berbeda. Pendekatan ini memungkinkan skalabilitas horizontal untuk operasi tulis, karena beban tersebar ke banyak node secara paralel.

Namun, sharding menambah kompleksitas desain. Operasi yang melibatkan data lintas shard memerlukan koordinasi tambahan, dan pemilihan kunci shard yang tidak tepat dapat menyebabkan ketidakseimbangan beban (*hot shard*) (Stonebraker et al., 2010).

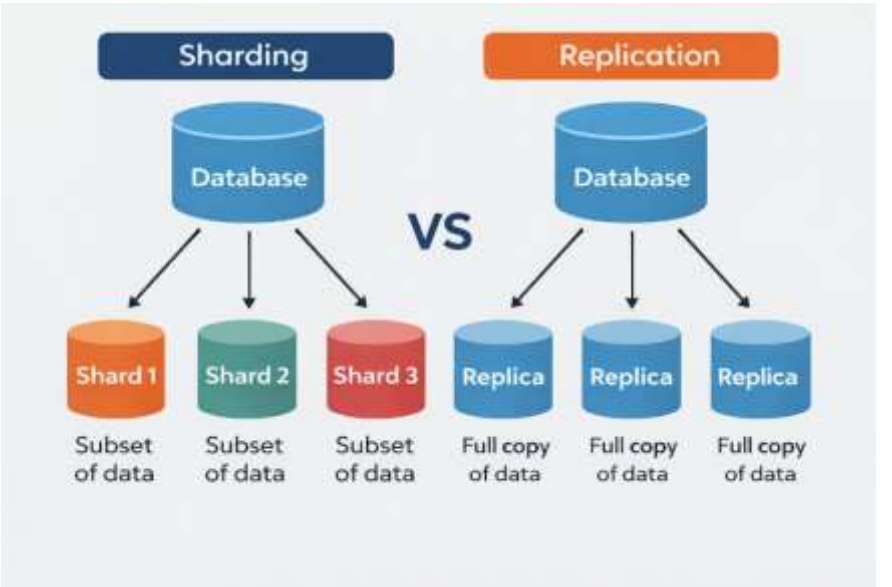
Contoh nyata: aplikasi e-commerce membagi data pengguna berdasarkan wilayah geografis agar transaksi dapat diproses secara paralel di pusat data terdekat.

Perbandingan Replikasi dan Sharding

Secara konseptual, perbedaan utama keduanya dapat diringkas sebagai berikut:

- Replikasi meningkatkan kapasitas baca dan ketersediaan, tetapi tidak meningkatkan kapasitas tulis secara signifikan.
- Sharding meningkatkan kapasitas tulis dan ukuran data yang dapat dikelola, tetapi menambah kompleksitas kueri dan pengelolaan.

Dalam praktik, banyak sistem besar menggunakan kombinasi replikasi dan sharding, di mana setiap shard direplikasi untuk mencapai kinerja dan keandalan yang seimbang (Kleppmann, 2017).



Gambar 15. Perbandingan mekanisme sharding dan replication pada arsitektur basis data terdistribusi.

LinkedIn-Pratima Upadhyay

Gambar tersebut menunjukkan perbedaan inti antara sharding dan replication dalam basis data terdistribusi: sharding membagi data ke beberapa shard berbeda yang ditempatkan di node terpisah sehingga beban penyimpanan dan tulis tersebar secara horizontal, meningkatkan kapasitas sistem tetapi membutuhkan mekanisme tambahan untuk menangani kueri lintas shard, sedangkan replication membuat setiap node menyimpan salinan data yang sama untuk meningkatkan ketersediaan dan kapasitas baca—jika satu node gagal, node lain tetap melayani—namun tidak banyak menambah kapasitas tulis dan memerlukan sinkronisasi demi konsistensi. Intinya, sharding mengejar skalabilitas dan kinerja tulis, sementara replication mengejar ketersediaan dan keandalan.

D. Message Queues vs Event Streams dalam Perancangan Sistem Terdistribusi

Dalam sistem terdistribusi modern, komunikasi asinkron dan pemrosesan data real-time menjadi kebutuhan utama untuk mencapai skalabilitas, ketahanan, dan responsivitas. Dua pola arsitektural yang paling banyak digunakan untuk tujuan ini adalah message queues dan event streams. Meskipun keduanya sama-sama mendukung komunikasi asinkron, keduanya dirancang untuk tujuan sistem yang berbeda dan membawa konsekuensi arsitektural yang tidak sama.

Message Queues

Message queues merupakan infrastruktur komunikasi yang memungkinkan pengiriman pesan secara asinkron melalui mekanisme antrian. Pesan dikirim oleh *producer* dan disimpan sementara hingga

diproses oleh *consumer*. Setelah pesan dikonsumsi, pesan tersebut umumnya dihapus dari antrian.

Karakteristik utama message queues meliputi:

- Decoupling temporal, di mana pengirim dan penerima tidak harus aktif secara bersamaan.
- Durabilitas pesan, pesan tetap tersimpan hingga berhasil diproses.
- Distribusi beban, satu pesan biasanya diproses oleh satu konsumen, sehingga cocok untuk *task distribution*.

Message queues sangat efektif untuk skenario yang berorientasi pada eksekusi tugas, bukan aliran data berkelanjutan.

Event Streams

Berbeda dengan message queues, event streams memandang data sebagai aliran peristiwa berurutan dan tidak dapat diubah (immutable log). Setiap event merepresentasikan perubahan keadaan sistem dan dapat dikonsumsi oleh banyak konsumen secara independen, baik secara real-time maupun dengan mekanisme *replay*.

Karakteristik utama event streams meliputi:

- Pemrosesan real-time dan berkelanjutan.
- Persistensi event jangka panjang, memungkinkan analisis historis dan rekonstruksi keadaan sistem.
- Dukungan multi-consumer, di mana satu event dapat diproses oleh banyak layanan.

Pendekatan ini sangat sesuai untuk sistem **event-driven**, analitik real-time, dan *event sourcing*.

Perbandingan Konseptual

Aspek	Message Queues	Event Streams
Tujuan utama	Distribusi tugas	Aliran data dan reaksi terhadap peristiwa
Model konsumsi	Satu pesan → satu konsumen	Satu event → banyak konsumen
Persistensi	Hingga pesan diproses	Log event disimpan jangka panjang
Replay data	Tidak umum	Didukung secara native
Fokus arsitektur	Keandalan eksekusi	Skalabilitas dan observabilitas data
Pola sistem	Task-oriented	Event-driven

Perbandingan ini menunjukkan bahwa message queues dan event streams bukan teknologi yang saling menggantikan, melainkan melengkapi tergantung pada kebutuhan sistem.

Use Case Utama

Message queues paling tepat digunakan untuk:

- Pemrosesan *background jobs*.
- Penjadwalan dan eksekusi tugas asinkron.
- Komunikasi antar microservices berbasis perintah (*command-based*).

Sebaliknya, event streams unggul dalam:

- Analitik real-time dan monitoring.
- Event sourcing dan audit trail.
- Deteksi anomali dan fraud secara langsung.
- Integrasi sistem berbasis data aliran.

Implikasi Arsitektural

Pemilihan antara message queues dan event streams berdampak langsung pada:

- Desain konsistensi data,
- Strategi skalabilitas,
- Kemampuan observability dan audit,
- serta kompleksitas operasional sistem.

Dalam praktik industri, banyak sistem besar mengombinasikan keduanya: message queues untuk orkestrasi tugas dan event streams untuk aliran data dan analitik.

Message queues dan event streams merepresentasikan dua paradigma berbeda dalam komunikasi asinkron. Memahami perbedaan konseptual dan implikasi desain keduanya memungkinkan arsitek sistem membangun solusi yang lebih tepat, efisien, dan berkelanjutan dalam lingkungan sistem terdistribusi modern.

E. Arsitektur Big Data

Pertumbuhan sistem digital modern menghasilkan data dalam jumlah yang sangat besar, cepat, dan beragam. Data ini tidak lagi dapat dikelola secara efektif menggunakan arsitektur basis data konvensional. Kondisi tersebut melahirkan konsep big data, yang merujuk pada pengelolaan dan pemrosesan data berskala sangat besar dengan karakteristik utama volume, velocity, dan variety (Chen, Mao, & Liu, 2014). Arsitektur big data dirancang untuk memungkinkan penyimpanan, pemrosesan, dan analisis data secara terdistribusi dengan kinerja tinggi dan toleransi terhadap kegagalan.

Karakteristik Utama Big Data

Big data umumnya ditandai oleh beberapa karakteristik berikut:

- **Volume**, yaitu ukuran data yang sangat besar.
- **Velocity**, yaitu kecepatan data dihasilkan dan diproses.
- **Variety**, yaitu keberagaman format data, baik terstruktur maupun tidak terstruktur.
- **Veracity**, yaitu kualitas dan keandalan data.
- **Value**, yaitu nilai informasi yang dapat diekstraksi dari data.

Karakteristik ini menuntut pendekatan arsitektural yang berbeda dibandingkan sistem data tradisional (Kleppmann, 2017).

Lapisan Arsitektur Big Data

Arsitektur big data umumnya terdiri atas beberapa lapisan utama:

1. **Data ingestion layer**. Bertugas mengumpulkan data dari berbagai sumber seperti aplikasi, sensor, log, dan sistem eksternal.
2. **Data storage layer**. Menyediakan penyimpanan terdistribusi yang mampu menangani data dalam skala besar dan beragam format.
3. **Data processing layer**. Mengolah data baik secara batch maupun real-time untuk menghasilkan informasi bermakna.
4. **Analytics dan visualization layer**. Menyajikan hasil analisis dalam bentuk laporan, dashboard, atau input untuk sistem lain.

Pembagian lapisan ini membantu memisahkan tanggung jawab dan meningkatkan skalabilitas sistem.

Batch Processing dan Stream Processing

Arsitektur big data mendukung dua pola pemrosesan utama. Batch processing digunakan untuk analisis data historis dalam jumlah besar dengan toleransi terhadap latensi. Sebaliknya, stream processing memungkinkan pemrosesan data secara langsung saat data dihasilkan, sehingga cocok untuk kasus yang membutuhkan respons cepat.

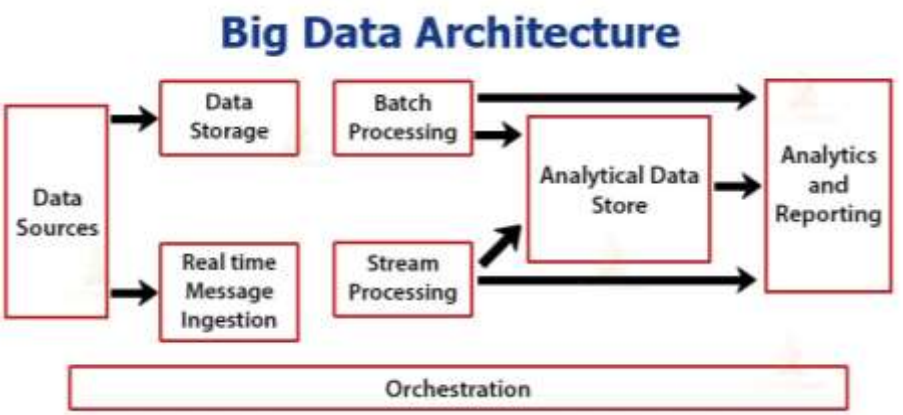
Banyak sistem modern mengombinasikan kedua pendekatan ini untuk memenuhi kebutuhan analitik jangka pendek dan jangka panjang secara bersamaan (Stonebraker et al., 2018).

Keunggulan dan Tantangan Arsitektur Big Data

Keunggulan utama arsitektur big data terletak pada kemampuannya untuk:

- menangani data dalam skala besar secara paralel,
- menyediakan toleransi terhadap kegagalan,
- serta mendukung analitik kompleks dan prediktif.

Namun, tantangan yang muncul meliputi kompleksitas sistem, biaya infrastruktur, kualitas data, dan kebutuhan keahlian teknis yang tinggi. Tanpa tata kelola yang baik, sistem big data berpotensi menghasilkan data yang besar tetapi bernilai rendah.



Gambar 16. Komponen utama arsitektur big data.

Arsitektur yang ditampilkan menggambarkan ekosistem big data end-to-end yang menggabungkan batch processing dan stream processing secara terorkestrasi, dimulai dari berbagai sumber data (aplikasi, log, sensor, transaksi) yang mengalir melalui dua jalur utama: jalur real-time, di mana data di-ingest, diproses secara streaming, dan disimpan ke analytical store untuk dashboard dan alert, serta jalur penyimpanan historis yang diproses secara berkala melalui batch processing untuk analitik mendalam, pelaporan, dan pelatihan model. Di pusat arsitektur, machine learning memanfaatkan data batch dan streaming sekaligus, menunjukkan integrasi analitik prediktif, sementara seluruh pipeline dikendalikan oleh lapisan orchestration yang mengatur penjadwalan, dependensi, dan keandalan sistem. Lebih dari sekadar pilihan teknis, arsitektur ini merupakan keputusan strategis yang harus selaras dengan tujuan bisnis, karakteristik data, dan kebutuhan analitik organisasi.

F. Data Governance dalam Sistem Skala Besar

Data governance adalah seperangkat kebijakan, peran, proses, dan standar yang memastikan data akurat, konsisten, aman, dan patuh sepanjang siklus hidupnya. Pada sistem skala besar dan terdistribusi di mana data berasal dari banyak sumber, mengalir lintas layanan, dan diproses secara real-time data governance menjadi fondasi untuk menjaga kepercayaan terhadap data dan keputusan yang dihasilkan (Otto, 2011). Tujuan utamanya meliputi peningkatan kualitas data, penguatan keamanan dan privasi, serta kepatuhan terhadap regulasi.

Skala dan distribusi menambah kompleksitas governance. Tantangan umum mencakup:

- **Fragmentasi data** akibat replikasi, sharding, dan multi-cloud.
- **Inkonstistensi skema** karena evolusi layanan yang cepat.
- **Keterlacakan terbatas** pada arsitektur event-driven.
- **Risiko kepatuhan** ketika data lintas wilayah dan yurisdiksi.

Tanpa governance yang jelas, sistem berisiko menghasilkan data besar namun bernilai rendah atau tidak dapat dipercaya (Kleppmann, 2017).

Komponen Utama Data Governance

Implementasi data governance yang efektif biasanya mencakup:

1. **Kualitas data**. Aturan validasi, pembersihan, dan pemantauan kualitas untuk memastikan data layak digunakan.
2. **Manajemen metadata**. Pencatatan asal-usul (*lineage*), definisi, dan kepemilikan data agar mudah ditelusuri.
3. **Keamanan dan kontrol akses**. Mekanisme otorisasi berbasis peran untuk melindungi data sensitif.

4. **Kepatuhan dan audit**. Proses audit berkala untuk memenuhi standar dan regulasi yang berlaku.

Komponen-komponen ini bekerja lintas lapisan arsitektur big data dan sistem terdistribusi

Data Governance dan Privasi

Privasi menjadi isu sentral dalam sistem skala besar. Data governance harus memastikan bahwa pengumpulan, penyimpanan, dan pemrosesan data pribadi mengikuti prinsip minimisasi data, tujuan penggunaan yang jelas, dan perlindungan hak subjek data. Kebijakan ini penting terutama pada domain seperti keuangan, kesehatan, dan layanan publik.

Governance dalam Arsitektur Event-Driven dan Big Data

Pada arsitektur event-driven, data mengalir cepat dan dikonsumsi oleh banyak layanan. Governance perlu diterapkan pada level event schema, versioning, dan retention policy agar perubahan tidak merusak konsumen lain. Dalam arsitektur big data, governance juga mengatur retensi data, penghapusan terkontrol, dan penggunaan kembali data untuk analitik dan pembelajaran mesin.

Pendekatan ini memastikan inovasi tetap berjalan tanpa mengorbankan stabilitas dan kepatuhan.

Dalam konteks rekayasa perangkat lunak tingkat lanjut, governance menyatukan aspek teknis dan organisasi agar sistem data skala besar tetap berkelanjutan. Data governance memastikan bahwa arsitektur data terdistribusi tidak hanya skalabel dan cepat, tetapi juga tepercaya, aman, dan patuh.

BAB VII
KEAMANAN PADA ARSITEKTUR MODERN

Perkembangan arsitektur modern ditandai oleh sistem terdistribusi, microservices, aplikasi cloud-native, dan integrasi lintas platform—memang mempercepat inovasi dan meningkatkan skalabilitas, tetapi sekaligus memperluas permukaan serangan karena setiap layanan, API, container, dan jalur komunikasi bisa menjadi titik masuk ancaman, sehingga keamanan tidak bisa lagi bersifat reaktif dan harus diintegrasikan sejak desain hingga operasi. Bab ini mengulas paradigma keamanan modern untuk sistem berskala besar, mulai dari analisis ancaman pada sistem terdistribusi, mekanisme authentication dan authorization yang adaptif, penerapan Zero Trust Architecture dalam pengelolaan akses, keamanan API dan microservices sebagai komponen kritis, integrasi keamanan melalui DevSecOps dalam siklus pengembangan, serta perlindungan data dan privasi digital sebagai dasar kepatuhan, kepercayaan pengguna, dan keberlanjutan sistem di era transformasi digital.

A. Ancaman Keamanan pada Sistem Terdistribusi

Sistem terdistribusi meningkatkan kinerja, skalabilitas, dan ketersediaan dengan memisahkan komponen ke berbagai layanan dan lokasi. Namun, karakteristik ini juga memperluas permukaan serangan karena tidak adanya satu titik kontrol tunggal. Setiap layanan, API, dan jalur komunikasi jaringan berpotensi menjadi celah keamanan jika tidak dirancang dengan prinsip *security by design*.

Ancaman utama muncul pada komunikasi jaringan, seperti *man-in-the-middle attack*, penyadapan, dan manipulasi data. Tanpa enkripsi

dan autentikasi antar layanan yang kuat, data sensitif dapat disusupi selama transmisi, terutama pada arsitektur microservices yang sangat bergantung pada API. Selain itu, kompromi identitas dan akses menjadi risiko signifikan akibat pengelolaan kredensial yang lemah, kesalahan konfigurasi hak akses, dan penggunaan kredensial statis, yang memungkinkan *privilege escalation* dan *lateral movement* antar layanan.

Dari sisi ketersediaan, serangan DDoS dapat menyebabkan degradasi layanan hingga *cascading failure* jika satu komponen kritis terganggu. Ancaman lain berasal dari kerentanan perangkat lunak dan rantai pasok, karena sistem modern banyak bergantung pada pustaka pihak ketiga dan *container images* yang dapat menyimpan celah keamanan tersembunyi.

B. Authentication dan Authorization Modern

Dalam arsitektur perangkat lunak modern, khususnya pada sistem terdistribusi dan microservices, authentication dan authorization tidak lagi dapat mengandalkan pendekatan terpusat dan statis. Dinamika layanan, integrasi lintas platform, serta penggunaan identitas non-manusia seperti layanan dan API menuntut mekanisme pengelolaan identitas yang fleksibel, terotomasi, dan skalabel. Authentication berfokus pada verifikasi identitas entitas yang mengakses sistem, baik pengguna maupun layanan, dengan pendekatan modern seperti token-based authentication, multi-factor authentication, dan sertifikat digital. Pendekatan ini meningkatkan keamanan sekaligus mempermudah integrasi lintas layanan melalui identity provider terpusat (Hardt, 2012).

Setelah identitas terverifikasi, authorization menentukan hak akses dan tindakan yang diizinkan berdasarkan prinsip *least privilege*. Model seperti role-based access control dan attribute-based access control memungkinkan pengelolaan akses yang dinamis dan kontekstual sesuai kebutuhan operasional (Ferraiolo et al., 2016). Dalam arsitektur microservices, tantangan semakin kompleks karena komunikasi terjadi antar layanan, sehingga diperlukan mekanisme service identity, autentikasi mutual, dan delegasi token untuk mencegah akses tidak sah dan pergerakan lateral. Oleh karena itu, authentication dan authorization modern harus terintegrasi dengan proses pengembangan dan operasi melalui otomatisasi, rotasi kredensial, serta pemantauan berkelanjutan agar berfungsi tidak hanya sebagai gerbang akses, tetapi juga sebagai instrumen kontrol keamanan sistem secara menyeluruh.

C. Zero Trust Architecture

Perkembangan sistem terdistribusi, komputasi awan, dan *microservices* telah mengubah secara fundamental batasan jaringan tradisional. Model keamanan lama yang berasumsi bahwa entitas di dalam jaringan internal dapat dipercaya (*trusted internal network*) tidak lagi relevan ketika aplikasi, data, dan pengguna tersebar di berbagai lokasi dan platform. Dalam konteks inilah Zero Trust Architecture (ZTA) muncul sebagai paradigma keamanan modern yang menolak asumsi kepercayaan implisit dan menggantinya dengan prinsip verifikasi berkelanjutan.

Inti dari Zero Trust Architecture adalah prinsip “never trust, always verify”. Setiap permintaan akses baik berasal dari pengguna internal, eksternal, maupun layanan sistem harus diverifikasi secara eksplisit sebelum diberikan akses ke sumber daya. Tidak ada

perbedaan perlakuan antara entitas yang berada di dalam atau di luar jaringan; semua dianggap berpotensi tidak tepercaya hingga terbukti sebaliknya. Pendekatan ini secara langsung menanggapi ancaman *lateral movement*, di mana penyerang yang berhasil menembus satu komponen sistem dapat menyebar ke komponen lain tanpa hambatan berarti.

Zero Trust Architecture dibangun di atas beberapa prinsip utama.

1. Pertama, **verifikasi identitas secara kuat**, yang mencakup autentikasi berlapis dan penggunaan identitas yang terkelola dengan baik untuk pengguna maupun layanan.
2. Kedua, **akses berbasis konteks dan kebijakan**, di mana keputusan akses mempertimbangkan berbagai atribut seperti peran, lokasi, perangkat, dan kondisi risiko saat permintaan dibuat.
3. Ketiga, **pembatasan akses minimum (*least privilege*)**, sehingga setiap entitas hanya memiliki hak akses yang benar-benar diperlukan untuk menjalankan fungsinya. Prinsip ini mengurangi dampak apabila terjadi kompromi identitas.

Dalam arsitektur modern, Zero Trust sering diterapkan melalui kombinasi teknologi dan kebijakan, seperti *identity and access management* (IAM), *micro-segmentation*, serta *continuous monitoring*. Micro-segmentation memecah sistem menjadi unit-unit kecil yang terisolasi, sehingga kegagalan atau kompromi pada satu layanan tidak langsung berdampak pada layanan lain. Sementara itu, pemantauan berkelanjutan memungkinkan sistem mendeteksi pola

akses yang mencurigakan dan menyesuaikan kebijakan secara dinamis.

Penerapan Zero Trust Architecture juga membawa tantangan tersendiri. Kompleksitas desain meningkat karena setiap interaksi harus diamankan dan diawasi. Selain itu, organisasi perlu menyesuaikan proses operasional dan budaya keamanan agar selaras dengan prinsip Zero Trust. Meskipun demikian, pendekatan ini memberikan kerangka yang lebih realistis dan adaptif dalam menghadapi ancaman keamanan modern.

D. Keamanan API dan Microservices

Dalam arsitektur modern, API dan microservices menjadi tulang punggung komunikasi antar sistem. Fleksibel, skalabel, dan cepat dikembangkan tetapi ada harga yang dibayar: permukaan serangan menjadi jauh lebih luas.

Mengapa API dan Microservices Rentan

Berbeda dengan aplikasi monolitik, microservices memecah sistem menjadi banyak layanan kecil yang saling berkomunikasi melalui API. Setiap API adalah pintu masuk potensial bagi serangan.

Masalah utamanya:

- Banyak endpoint tidak terekspos lewat antarmuka pengguna
- Komunikasi berbasis data (JSON/XML) sulit divalidasi secara manual
- Dokumentasi API sering tidak lengkap atau tidak diperbarui

Akibatnya, celah keamanan bisa tersembunyi lama tanpa disadari.

Jenis Ancaman Umum

Beberapa risiko utama pada API dan microservices meliputi:

- Broken Authentication & Authorization. Token bocor, role salah konfigurasi, atau endpoint sensitif tanpa proteksi.
- Injection Attacks. SQL Injection, Command Injection, atau XML External Entity (XXE) melalui payload API.
- Excessive Data Exposure. API mengirim lebih banyak data daripada yang dibutuhkan klien.
- Insecure Service-to-Service Communication. Layanan internal diasumsikan "aman" padahal bisa dieksploitasi jika satu layanan ditembus.
- Lack of Rate Limiting. Membuka peluang brute-force dan denial-of-service.

Keamanan microservices tidak gagal karena satu bug besar, tetapi karena banyak celah kecil yang terakumulasi:

- Konsistensi keamanan antar layanan sulit dijaga
- Tim berbeda → standar keamanan berbeda
- API baru sering muncul tanpa audit keamanan
- Sulit mengetahui seluruh API yang aktif di sistem

Singkatnya: kompleksitas adalah musuh keamanan.

Prinsip Keamanan yang Wajib Diterapkan

Pendekatan efektif bukan reaktif, tapi struktural:

1. Security by Design. Keamanan dirancang sejak tahap arsitektur, bukan ditambal di akhir.

2. Strong Authentication & Authorization. Gunakan OAuth2, JWT dengan validasi ketat, dan prinsip least privilege.
3. Validasi Input yang Ketat. Jangan percaya data dari klien, bahkan dari layanan internal.
4. Enkripsi Komunikasi. HTTPS/TLS wajib, termasuk komunikasi antar microservices.
5. API Inventory & Documentation. Semua API harus terdokumentasi dan diaudit secara berkala.

Pengujian Keamanan API

Pengujian API berbeda dari web tradisional karena tidak bergantung pada UI.

Pendekatan efektif:

- *Automated scanning* berbasis API specification (OpenAPI/Swagger)
- *Manual testing* untuk logika bisnis dan authorization
- *Continuous testing* seiring perubahan layanan

Tanpa definisi API yang jelas, pengujian keamanan akan selalu timpang.

API dan microservices bukan “lebih berbahaya” secara bawaan—tetapi lebih mudah lalai. Ketika sistem terfragmentasi, tanggung jawab keamanan ikut terfragmentasi.

Jika keamanan tidak menjadi budaya teknis, maka: sistem akan tetap berjalan… sampai suatu hari bocor.

E. DevSecOps

DevSecOps *(Development, Security, Operations)* merupakan pendekatan pengembangan perangkat lunak yang mengintegrasikan aspek keamanan ke dalam seluruh proses DevOps. Keamanan tidak lagi ditempatkan sebagai tahap akhir, melainkan menjadi bagian dari setiap tahapan pengembangan sistem.

Pendekatan ini berkembang karena sistem modern menuntut kecepatan, sementara model keamanan tradisional bersifat lambat dan reaktif.

Pada praktik konvensional, pengujian keamanan sering dilakukan setelah sistem selesai dikembangkan. Kondisi ini menyebabkan kerentanan baru diketahui ketika sistem sudah digunakan.

Beberapa faktor yang melatarbelakangi penerapan DevSecOps adalah:

- Percepatan siklus pengembangan perangkat lunak
- Peningkatan kompleksitas sistem berbasis cloud dan microservices
- Tingginya biaya perbaikan jika kerentanan ditemukan terlambat

DevSecOps hadir untuk memastikan keamanan berjalan seiring dengan proses pengembangan.

Prinsip Dasar DevSecOps

DevSecOps menekankan bahwa keamanan merupakan tanggung jawab bersama.

Prinsip utamanya meliputi:

- Keamanan diterapkan sejak tahap awal pengembangan
- Proses keamanan dilakukan secara otomatis
- Pengujian keamanan dilakukan secara berkelanjutan
- Seluruh tim terlibat dalam menjaga keamanan sistem

Penerapan DevSecOps dalam Siklus Pengembangan

DevSecOps diterapkan pada setiap tahap siklus pengembangan perangkat lunak, yaitu:

- **Perancangan**: identifikasi potensi ancaman dan perancangan sistem yang aman
- **Pengembangan**: pemeriksaan kode sumber dan dependensi
- **Pengujian**: pengujian keamanan aplikasi dan API
- **Penerapan**: pemeriksaan konfigurasi dan pengelolaan kredensial
- **Operasional**: pemantauan sistem dan penanganan insiden

Dengan pendekatan ini, risiko keamanan dapat dikendalikan sejak dini.

DevSecOps dalam Sistem API dan Microservices

Pada sistem berbasis API dan microservices, perubahan layanan terjadi secara cepat. DevSecOps membantu menjaga konsistensi keamanan pada setiap layanan.

Manfaat utama penerapan DevSecOps antara lain:

- Pengujian API dilakukan secara otomatis
- Standar keamanan diterapkan secara konsisten

- Kerentanan dapat dideteksi lebih awal

DevSecOps tidak bertujuan menghilangkan seluruh risiko keamanan, tetapi memastikan risiko tersebut dapat dikelola secara sistematis. Dalam pengembangan sistem modern, integrasi keamanan sejak awal merupakan langkah penting untuk menjaga keandalan dan keberlanjutan sistem.

F. Proteksi Data dan Privasi Digital

Dalam arsitektur perangkat lunak modern berbasis cloud dan sistem terdistribusi, data menjadi aset strategis sekaligus sumber risiko utama karena implikasi teknis, hukum, etika, dan reputasi yang menyertainya. Oleh sebab itu, proteksi data dan privasi digital harus terintegrasi langsung dalam desain arsitektur, proses operasional, dan tata kelola sistem. Proteksi data berfokus pada menjaga kerahasiaan, integritas, dan ketersediaan data sepanjang siklus hidupnya, terutama dalam lingkungan terdistribusi yang melibatkan replikasi dan transmisi lintas jaringan serta wilayah geografis. Sementara itu, privasi digital menekankan hak individu atas data pribadi melalui prinsip transparansi, pembatasan tujuan, dan minimisasi data agar hanya informasi yang diperlukan yang diproses dan disimpan. Mekanisme seperti enkripsi data saat disimpan dan ditransmisikan, pengelolaan kunci yang aman, kontrol akses berbasis *least privilege*, serta audit log yang andal menjadi fondasi penting untuk membatasi, memantau, dan melindungi akses data dari kebocoran maupun penyalahgunaan.

BAB VIII
OBSERVABILITY, MONITORING, DAN RELIABILITY ENGINEERING

Bab ini membahas fondasi dan praktik utama dalam menjaga keandalan (*reliability*) dan ketahanan (*resilience*) sistem perangkat lunak modern. Pembahasan dimulai dari konsep observability dan perannya dalam sistem terdistribusi, dilanjutkan dengan logging terdistribusi serta monitoring kinerja dan infrastruktur. Selanjutnya, Bab ini mengkaji pendekatan *Site Reliability Engineering* (SRE) sebagai disiplin yang mengintegrasikan rekayasa perangkat lunak dan operasi sistem. Konsep *Chaos Engineering* dibahas untuk memahami dan meningkatkan ketahanan sistem melalui eksperimen terkontrol terhadap kegagalan. Baab ini ditutup dengan pembahasan manajemen insiden dan resiliensi sistem sebagai upaya sistematis untuk memulihkan layanan dan menjaga kontinuitas operasional. Melalui Bab ini, pembaca diharapkan memperoleh kerangka berpikir dan praktik yang komprehensif dalam merancang sistem yang tidak hanya berfungsi, tetapi juga andal dan berkelanjutan dalam menghadapi ketidakpastian.

A. Konsep Observability Sistem

Observability sistem adalah kemampuan untuk memahami kondisi internal suatu sistem berdasarkan data yang dihasilkan oleh sistem tersebut. Data ini digunakan untuk mengetahui bagaimana sistem bekerja, mendeteksi gangguan, serta menganalisis penyebab masalah secara akurat.

Dalam sistem modern yang bersifat kompleks dan terdistribusi, observability menjadi sangat penting karena perilaku sistem tidak lagi dapat dipahami hanya dari satu komponen atau satu indikator saja.

Perkembangan arsitektur berbasis cloud, container, dan microservices menyebabkan sistem menjadi lebih dinamis dan sulit dipantau secara konvensional. Gangguan pada satu layanan dapat berdampak pada layanan lain, sehingga penyebab masalah sering tidak terlihat secara langsung.

Observability hadir untuk memberikan visibilitas menyeluruh terhadap perilaku sistem, bukan hanya mendeteksi bahwa suatu masalah terjadi, tetapi juga menjelaskan mengapa dan di mana masalah tersebut muncul.

Tujuan Observability Sistem

Tujuan utama observability sistem adalah:

- Memahami kondisi dan perilaku sistem secara menyeluruh
- Mempercepat proses identifikasi dan analisis masalah
- Mendukung pengambilan keputusan berbasis data
- Meningkatkan keandalan dan stabilitas sistem

Dengan observability yang baik, tim dapat merespons gangguan dengan lebih cepat dan tepat.

Komponen Utama Observability

Observability sistem dibangun dari beberapa jenis data utama, yaitu:

- **Metrics**. Data numerik yang menunjukkan kondisi sistem, seperti penggunaan CPU, memori, dan waktu respons.
- **Logs**. Catatan peristiwa yang terjadi di dalam sistem, termasuk pesan kesalahan dan aktivitas aplikasi.
- **Traces**. Informasi alur permintaan yang melintasi beberapa layanan, berguna untuk sistem terdistribusi.
- **Events**. Informasi kejadian tertentu, seperti perubahan konfigurasi atau proses deployment.

Kombinasi data ini memungkinkan analisis yang lebih mendalam terhadap perilaku sistem.

Observability dan Monitoring

Observability sering disamakan dengan monitoring, namun keduanya memiliki perbedaan mendasar.

Monitoring berfokus pada pemantauan kondisi tertentu berdasarkan indikator yang telah ditentukan sebelumnya. Sementara itu, observability memungkinkan analisis masalah yang belum diprediksi, karena menyediakan data yang cukup untuk memahami sistem secara menyeluruh.

Dengan kata lain, monitoring menjawab *apa yang terjadi*, sedangkan observability menjawab *mengapa hal tersebut terjadi*.

Peran Observability dalam Sistem Modern

Dalam pengelolaan sistem modern, observability berperan sebagai dasar untuk:

- Pemeliharaan sistem yang proaktif
- Optimalisasi kinerja aplikasi
- Deteksi dini gangguan dan anomali
- Peningkatan kualitas layanan secara berkelanjutan

Observability bukan hanya alat teknis, tetapi bagian penting dari strategi pengelolaan sistem yang andal.

B. Logging Terdistribusi

Logging terdistribusi adalah proses pencatatan peristiwa atau kejadian (*logging*) yang terjadi pada berbagai komponen dalam sistem perangkat lunak yang terdistribusi seperti arsitektur microservices atau cluster server. Sistem ini menghasilkan log dari banyak layanan yang berjalan pada mesin, kontainer, atau node yang berbeda, kemudian log-log tersebut dikumpulkan dan dikelola secara terpusat sehingga memudahkan analisis dan pemantauan secara keseluruhan (*distributed logging*).

Tujuan dan Peran Logging Terdistribusi

1. **Visibilitas Sistem Secara Menyeluruh**. Logging terdistribusi memungkinkan tim pengembang dan operasional melihat berbagai aktivitas di seluruh layanan dalam satu tampilan terpadu, tidak terpisah per layanan. Ini penting untuk memahami alur kejadian dan perilaku sistem secara keseluruhan.

2. **Pemecahan Masalah yang Lebih Efisien**. Ketika satu permintaan (*request*) melewati beberapa layanan, log terdistribusi membantu menghubungkan seluruh jejak dari permintaan tersebut melalui berbagai layanan, sehingga memudahkan identifikasi sumber masalah dan waktu kejadian secara akurat.

3. **Dukungan Observability dan Pemantauan**. Dalam konteks observability, logging terdistribusi menjadi pilar penting karena log memberikan catatan waktu dan kejadian yang terjadi di berbagai layanan, yang kemudian dapat dianalisis bersama dengan metrik dan jejak (*traces*) untuk memberikan pemahaman yang lebih kaya tentang kondisi sistem.

Proses Logging Terdistribusi

Secara garis besar, proses logging terdistribusi terdiri atas beberapa tahapan:

1. **Pencatatan (Log Generation)**. Setiap layanan dalam sistem mencatat kejadian penting seperti error, status operasi, atau aktivitas yang terjadi di dalamnya.

2. **Pengumpulan (Log Collection)**. Log yang dihasilkan dari berbagai layanan dikumpulkan secara otomatis melalui *agent* atau *collector* yang berjalan di setiap host atau kontainer, yang kemudian diteruskan ke pusat pengelolaan log.

3. **Penggabungan (Log Aggregation)**. Log-log yang telah dikumpulkan disatukan dalam satu repositori atau sistem pengelolaan log sehingga dapat dianalisis secara keseluruhan tanpa kehilangan konteks.

4. **Penyimpanan dan Analisis**. Setelah terpusat, log dapat diindeks dan dianalisis menggunakan alat seperti Elasticsearch atau sistem dashboard seperti Kibana dan Grafana untuk pencarian, visualisasi, dan pembuatan peringatan (*alert*).

Tantangan dalam Logging Terdistribusi

Implementasi logging terdistribusi bukan tanpa tantangan. Beberapa isu yang sering muncul antara lain:

- **Volume Data Log yang Sangat Besar**. Sistem terdistribusi menghasilkan banyak log dari berbagai sumber, yang memerlukan ruang penyimpanan dan kapasitas pemrosesan yang besar.
- **Konsistensi Waktu (Time Synchronization)**. Tanpa sinkronisasi waktu yang baik di seluruh node, log dari berbagai layanan bisa menunjukkan urutan kejadian yang tidak akurat, sehingga menyulitkan analisis.
- **Pengelolaan Konteks Lintas Layanan**. Untuk mengetahui alur lengkap operasi, diperlukan kolerasi log menggunakan *correlation ID* atau sejenisnya, agar setiap log terkait dengan permintaan yang sama dapat dihubungkan.

Logging terdistribusi merupakan bagian krusial dari strategi observability dalam sistem modern. Tanpa adanya logging yang terkoordinasi, tim akan kesulitan dalam:

- Mendiagnosis kegagalan yang melibatkan beberapa layanan
- Mengetahui jejak lengkap dari suatu kejadian

- Menyediakan informasi kontekstual bagi sistem pemantauan dan pelacakan terdistribusi (*distributed tracing*) yang lebih maju.

Dengan demikian, logging terdistribusi bukan hanya sekadar pencatatan peristiwa, tetapi juga fondasi untuk analisis yang cepat, akurat, serta integrasi dengan metrik dan jejak dalam rangka meningkatkan ketahanan dan performa sistem.

C. Monitoring Kinerja dan Infrastruktur

Monitoring kinerja dan infrastruktur merupakan bagian inti dari observability sistem yang berfokus pada pengamatan kondisi sumber daya secara real-time dan historis. Tujuan utama monitoring adalah memastikan bahwa sistem berjalan dalam kondisi normal, stabil, dan memiliki kapasitas yang cukup untuk melayani beban kerja.

Pada arsitektur modern, monitoring umumnya dilakukan menggunakan Prometheus sebagai sistem pengumpul metrik dan Grafana sebagai alat visualisasi.

Prometheus berfungsi mengumpulkan metrik dari sistem melalui komponen yang disebut *exporter*. Dalam konteks monitoring infrastruktur, Node Exporter digunakan untuk mengekspose metrik tingkat sistem operasi, seperti:

- Penggunaan CPU
- Konsumsi memori
- Kapasitas dan penggunaan disk
- Beban sistem (*system load*)
- Waktu aktif sistem (*uptime*)

Metrik-metrik ini dikumpulkan secara berkala dan disimpan sebagai data *time-series*, sehingga memungkinkan analisis perubahan kondisi sistem dari waktu ke waktu.

Visualisasi Monitoring dengan Grafana

Gambar 17. Dashboard Monitoring Kinerja dan Infrastruktur Sistem menggunakan Prometheus dan Grafana (Node Exporter)

Grafana digunakan untuk menampilkan data yang dikumpulkan Prometheus dalam bentuk dashboard visual yang mudah dipahami. Dashboard seperti pada gambar menampilkan kondisi sistem secara ringkas melalui indikator (*gauge*), grafik, dan nilai numerik.

Beberapa komponen utama yang dimonitor antara lain:

a. Monitoring CPU

- CPU Busy menunjukkan persentase waktu CPU sedang digunakan.
- System Load (1 menit dan 5 menit) menggambarkan tekanan kerja terhadap CPU dalam interval waktu tertentu.
- Informasi jumlah CPU **core** membantu memahami kapasitas komputasi sistem.

Nilai CPU yang rendah menunjukkan sistem dalam kondisi ringan dan tidak mengalami beban berlebih.

b. Monitoring Memori

- Total RAM, RAM Used, dan RAM Free menunjukkan distribusi penggunaan memori.
- Cache dan Buffer menunjukkan memori yang digunakan sistem operasi untuk optimasi kinerja.
- SWAP Used menunjukkan apakah sistem kekurangan RAM dan harus menggunakan disk sebagai memori cadangan.

Penting untuk dicatat bahwa rendahnya memori bebas tidak selalu berarti masalah, karena sistem operasi secara aktif memanfaatkan memori untuk cache.

c. Monitoring Disk

- Root File System Usage menunjukkan persentase ruang disk yang telah digunakan.
- Informasi Total RootFS memberikan gambaran kapasitas penyimpanan yang tersedia.

Monitoring disk penting untuk mencegah kegagalan sistem akibat kehabisan ruang penyimpanan.

d. Monitoring Stabilitas Sistem

- Uptime menunjukkan lama waktu sistem berjalan sejak terakhir dinyalakan ulang.
- Data ini berguna untuk mengidentifikasi restart yang tidak direncanakan atau gangguan sistem.

Manfaat Monitoring Kinerja dan Infrastruktur

Monitoring menggunakan Prometheus dan Grafana memberikan beberapa manfaat utama, antara lain:

1. Deteksi dini masalah sebelum berdampak pada layanan.
2. Pemahaman kapasitas sistem, apakah perlu peningkatan sumber daya atau tidak.
3. Analisis performa historis untuk evaluasi dan perencanaan.
4. Dasar pengambilan keputusan teknis berbasis data aktual, bukan asumsi.

Posisi Monitoring dalam Observability

Monitoring kinerja dan infrastruktur menjawab pertanyaan “apa yang sedang terjadi pada sistem”, tetapi belum sepenuhnya menjawab “mengapa hal tersebut terjadi”. Oleh karena itu, monitoring perlu dilengkapi dengan:

- Logging terdistribusi
- Tracing antar layanan

Namun demikian, monitoring infrastruktur tetap menjadi fondasi utama observability, karena tanpa kondisi sistem yang sehat, analisis tingkat aplikasi tidak akan optimal.

D. Site Reliability Engineering (SRE)

Site Reliability Engineering (SRE) adalah pendekatan rekayasa yang bertujuan menjaga keandalan sistem perangkat lunak skala besar dengan menerapkan prinsip rekayasa perangkat lunak pada aktivitas operasi sistem. Pendekatan ini muncul seiring meningkatnya kompleksitas sistem berbasis *cloud*, *microservices*, dan infrastruktur terdistribusi, di mana metode operasi tradisional yang manual dan reaktif tidak lagi mampu menjamin stabilitas layanan.

Dalam SRE, keandalan diperlakukan sebagai masalah teknis yang dapat direkayasa, bukan sekadar tanggung jawab operasional. Artinya, stabilitas sistem dicapai melalui desain, automasi, pengukuran, dan perbaikan berkelanjutan.

Perbedaan SRE dan Operasi TI Tradisional

Pendekatan operasi tradisional berfokus pada menjaga sistem tetap berjalan dan meminimalkan perubahan. Sebaliknya, SRE mengakui bahwa perubahan adalah keniscayaan dalam pengembangan perangkat lunak modern.

Perbedaan utama:

- Operasi tradisional → stabilitas diutamakan, perubahan dibatasi
- SRE → stabilitas dan inovasi diseimbangkan
- Operasi tradisional → banyak pekerjaan manual
- SRE → automasi sebagai prinsip utama

Dengan pendekatan ini, SRE memungkinkan organisasi tetap berinovasi tanpa mengorbankan keandalan layanan.

Service Level Indicators (SLI) dan Service Level Objectives (SLO)

Fondasi pengukuran dalam SRE adalah Service Level Indicators (SLI), yaitu metrik kuantitatif yang menggambarkan kualitas layanan, seperti:

- waktu respons (*latency*),
- tingkat ketersediaan (*availability*),
- tingkat kesalahan (*error rate*).

Berdasarkan SLI, organisasi menetapkan Service Level Objectives (SLO), yaitu target numerik yang mendefinisikan tingkat layanan yang diharapkan oleh pengguna. Dengan SLO, keandalan sistem tidak lagi bersifat subjektif, tetapi dapat diukur dan dievaluasi secara objektif.

Salah satu konsep paling khas dalam SRE adalah error budget, yaitu batas kegagalan yang masih dapat ditoleransi berdasarkan SLO. Error budget merepresentasikan "ruang kegagalan" yang diizinkan dalam suatu periode tertentu.

Selama error budget masih tersedia:

- tim pengembang dapat melakukan rilis fitur baru,
- eksperimen dan inovasi tetap berjalan.

Namun, ketika error budget habis:

- prioritas dialihkan ke peningkatan stabilitas,
- perubahan sistem dibatasi hingga keandalan pulih.

Pendekatan ini menciptakan mekanisme pengambilan keputusan yang objektif antara kebutuhan bisnis dan keandalan teknis.

Automasi dan Eliminating Toil

SRE menekankan pentingnya automasi dalam operasi sistem. Aktivitas operasional yang bersifat berulang, manual, dan tidak memberikan nilai jangka panjang disebut sebagai *toil*.

Contoh *toil*:

- restart layanan secara manual,
- deployment berulang tanpa automasi,
- respons insiden yang tidak terdokumentasi.

Melalui automasi, SRE berupaya mengurangi *toil* sehingga tim dapat fokus pada:

- peningkatan desain sistem,
- perbaikan keandalan jangka panjang,
- pengembangan alat bantu internal.

Peran SRE dalam Sistem Modern

Dalam arsitektur perangkat lunak modern, SRE berperan sebagai jembatan antara pengembangan dan operasi. SRE memastikan bahwa sistem tidak hanya cepat dikembangkan, tetapi juga:

- stabil di lingkungan produksi,
- mudah dipantau dan dianalisis,
- mampu pulih dengan cepat saat terjadi kegagalan.

Dengan demikian, Site Reliability Engineering menjadi fondasi penting bagi sistem yang andal, adaptif, dan berkelanjutan, sekaligus

menjadi landasan konseptual bagi pembahasan selanjutnya mengenai Chaos Engineering sebagai pendekatan pengujian ketahanan sistem.

E. Chaos Engineering

Chaos Engineering adalah pendekatan eksperimental yang bertujuan meningkatkan ketahanan (*resilience*) sistem perangkat lunak dengan cara menguji sistem melalui gangguan yang disengaja dan terkontrol. Pendekatan ini berangkat dari realitas bahwa dalam sistem terdistribusi modern, kegagalan tidak dapat sepenuhnya dihindari. Gangguan jaringan, kegagalan layanan, lonjakan trafik, atau kesalahan konfigurasi merupakan bagian normal dari operasi sistem skala besar. Oleh karena itu, fokus Chaos Engineering bukan pada pencegahan kegagalan total, melainkan pada kemampuan sistem untuk tetap berfungsi dan pulih ketika kegagalan terjadi.

Tujuan utama Chaos Engineering adalah membangun kepercayaan terhadap sistem dengan cara membuktikan bahwa sistem mampu bertahan dalam kondisi tidak ideal. Dengan melakukan eksperimen kegagalan secara terencana, tim dapat menemukan titik lemah sistem sebelum kegagalan tersebut muncul secara nyata di lingkungan produksi.

Chaos Engineering didasarkan pada beberapa prinsip utama. Pertama, eksperimen harus dimulai dari kondisi normal sistem, sehingga dampak gangguan dapat dibandingkan secara jelas. Kedua, eksperimen dilakukan secara terkontrol dan terukur, dengan cakupan yang dibatasi agar risiko terhadap pengguna tetap minimal. Ketiga, hasil eksperimen harus dianalisis secara sistematis untuk menghasilkan pembelajaran yang dapat ditindaklanjuti.

Prinsip ini membedakan Chaos Engineering dari pengujian acak tanpa tujuan. Chaos Engineering menempatkan kegagalan sebagai alat pembelajaran, bukan sebagai ancaman yang harus dihindari sepenuhnya.

Jenis Gangguan dalam Chaos Engineering

Eksperimen Chaos Engineering dapat mencakup berbagai jenis gangguan, tergantung pada arsitektur sistem dan tujuan pengujian. Contoh gangguan yang umum diuji antara lain:

- penghentian layanan secara tiba-tiba,
- peningkatan latensi jaringan,
- kegagalan sebagian infrastruktur,
- pembatasan sumber daya seperti CPU atau memori,
- kehilangan koneksi ke sistem eksternal.

Melalui variasi gangguan ini, tim dapat mengevaluasi bagaimana sistem bereaksi terhadap skenario kegagalan yang realistis dan kompleks.

Hubungan Chaos Engineering dan Observability

Chaos Engineering sangat bergantung pada observability yang baik. Tanpa data yang memadai dari logs, metrics, dan traces, dampak eksperimen kegagalan sulit dipahami secara menyeluruh. Observability memungkinkan tim untuk:

- memantau perubahan perilaku sistem selama eksperimen,
- mendeteksi efek samping yang tidak diharapkan,
- mengidentifikasi akar penyebab kegagalan dengan cepat.

Dengan demikian, Chaos Engineering dan observability saling melengkapi dalam membangun sistem yang transparan dan tangguh.

Peran Chaos Engineering dalam SRE

Dalam konteks Site Reliability Engineering (SRE), Chaos Engineering digunakan sebagai alat untuk memvalidasi asumsi keandalan sistem. SRE menetapkan target keandalan melalui SLO dan error budget, sedangkan Chaos Engineering menguji apakah sistem benar-benar mampu memenuhi target tersebut dalam kondisi ekstrem.

Pendekatan ini membantu tim SRE menghindari rasa aman semu (*false confidence*) terhadap sistem yang tampak stabil, tetapi belum pernah diuji dalam skenario kegagalan nyata.

Manfaat Jangka Panjang Chaos Engineering

Penerapan Chaos Engineering secara konsisten memberikan manfaat jangka panjang bagi organisasi. Tim menjadi lebih siap menghadapi insiden, sistem dirancang dengan mempertimbangkan kegagalan sejak awal, dan budaya organisasi bergeser dari reaktif menjadi proaktif. Dengan demikian, Chaos Engineering bukan sekadar teknik pengujian, melainkan strategi pembelajaran berkelanjutan untuk meningkatkan ketahanan sistem perangkat lunak modern.

Contoh Chaos Engineering

1. Mematikan Server Secara Sengaja (Server Failure Injection)

Contoh kasus: Pada sistem e-commerce berbasis microservices, satu instance Order Service sengaja dimatikan saat jam sibuk.

Tujuan uji:

- Apakah load balancer otomatis mengalihkan trafik?
- Apakah pengguna tetap bisa checkout?
- Apakah sistem recovery berjalan otomatis?

Pelajaran utama: Sistem *tidak boleh bergantung pada satu server*.

2. Simulasi Koneksi Jaringan Lambat atau Terputus

Contoh kasus: Latency buatan ditambahkan antara Payment Service dan Database.

Tujuan uji:

- Apakah sistem timeout dengan benar?
- Apakah retry mechanism bekerja?
- Apakah pengguna mendapat pesan yang jelas?

Pelajaran utama: Jaringan adalah sumber kegagalan paling umum di sistem terdistribusi.

3. Chaos pada Database (Read/Write Failure)

Contoh kasus: Akses write ke satu shard database dinonaktifkan sementara.

Tujuan uji:

- Apakah sistem fallback ke replica?
- Apakah transaksi gagal secara konsisten (bukan sebagian)?
- Apakah data tetap konsisten?

Pelajaran utama: Chaos membantu menguji ketahanan data, bukan hanya layanan.

4. Lonjakan Trafik Ekstrem (Traffic Spike Chaos)

Contoh kasus: Sistem diuji dengan trafik 10× lipat seperti flash sale.

Tujuan uji:

- Apakah auto-scaling berjalan?
- Apakah cache bekerja efektif?
- Apakah sistem tetap responsif?

Pelajaran utama: Skalabilitas tidak bisa diasumsikan, harus dibuktikan.

5. Kegagalan Dependency Eksternal

Contoh kasus: API pihak ketiga (payment gateway / shipping API) disimulasikan down.

Tujuan uji:

- Apakah circuit breaker aktif?
- Apakah sistem tetap berjalan dalam mode terbatas?
- Apakah pengguna diberi informasi yang benar?

Pelajaran utama: Sistem harus *tahan terhadap kegagalan pihak luar*.

6. Chaos Engineering di Sistem Finansial

Contoh kasus fintech: Delay disengaja pada fraud detection service.

Tujuan uji:

- Apakah transaksi ditunda dengan aman?
- Apakah tidak terjadi double charge?
- Apakah audit trail tetap lengkap?

Pelajaran utama: Chaos Engineering tidak hanya soal uptime, tetapi soal trust.

7. Chaos pada Level Organisasi (GameDay)

Contoh kasus: Tim SRE menjalankan simulasi "server utama mati total" tanpa pemberitahuan.

Tujuan uji:

- Apakah tim merespons sesuai SOP?
- Apakah komunikasi berjalan efektif?
- Apakah keputusan teknis tepat di bawah tekanan?

Pelajaran utama: Ketahanan sistem bergantung pada manusia dan proses, bukan teknologi saja.

Chaos Engineering bukan membuat sistem rusak, tetapi membuktikan bahwa sistem tetap hidup saat rusak.

Chaos Engineering membantu organisasi:

- Mengurangi risiko kegagalan besar

- Memvalidasi arsitektur terdistribusi
- Membangun *confidence* terhadap sistem produksi

F. Manajemen Insiden dan Resiliensi Sistem

Manajemen insiden dan resiliensi sistem merupakan bagian penting dalam menjaga keberlangsungan layanan pada sistem modern. Manajemen insiden berfokus pada penanganan gangguan secara terstruktur, sedangkan resiliensi sistem menekankan kemampuan sistem untuk tetap berfungsi, pulih dengan cepat, dan beradaptasi ketika terjadi kegagalan.

Keduanya saling terkait dan menjadi pelengkap dari pendekatan Site Reliability Engineering (SRE).

Insiden sistem adalah kejadian yang menyebabkan layanan:

- Tidak tersedia,
- Mengalami penurunan kinerja,
- Berperilaku tidak sesuai dengan Service Level Objective (SLO).

Contoh insiden antara lain lonjakan penggunaan CPU, kehabisan memori, disk penuh, atau restart sistem yang tidak direncanakan.

Deteksi Insiden Berbasis Monitoring

Dalam sistem yang dimonitor menggunakan Prometheus dan Grafana, insiden dapat dideteksi melalui perubahan metrik secara signifikan, misalnya:

- CPU usage meningkat tajam dalam waktu singkat.

- Penggunaan memori terus naik hingga mendekati batas maksimum.
- Ruang disk mencapai ambang kritis.
- System load meningkat melebihi kapasitas CPU.

Grafana digunakan untuk memantau kondisi tersebut secara visual, sementara Prometheus menyediakan data historis untuk analisis tren.

Contoh Alur Manajemen Insiden

Sebagai contoh konkret, sebuah server aplikasi menunjukkan peningkatan penggunaan memori secara terus-menerus hingga hampir mencapai kapasitas maksimum.

Langkah manajemen insiden yang dilakukan adalah:

1. Identifikasi. Dashboard Grafana menunjukkan anomali penggunaan memori.
2. Analisis awal. Data historis Prometheus digunakan untuk melihat pola peningkatan memori dari waktu ke waktu.
3. Mitigasi. Layanan direstart secara terkontrol atau skala sumber daya ditingkatkan sementara.
4. Pemulihan. Sistem kembali ke kondisi normal dan layanan dapat diakses pengguna.
5. Evaluasi pasca-insiden. Dilakukan analisis akar masalah (*post-incident review*) untuk mencegah kejadian serupa.

Pendekatan ini menekankan penanganan insiden yang cepat dan terukur.

Konsep Resiliensi Sistem

Resiliensi sistem adalah kemampuan sistem untuk:

- Menyerap gangguan,
- Tetap menyediakan layanan minimal,
- Pulih dalam waktu singkat setelah kegagalan.

Resiliensi tidak berarti sistem bebas dari kegagalan, melainkan mampu menghadapi kegagalan tanpa dampak besar bagi pengguna.

Strategi Meningkatkan Resiliensi Sistem

Beberapa strategi yang umum diterapkan antara lain:

1. Monitoring berkelanjutan. Memastikan kondisi sistem selalu terpantau.
2. Otomatisasi pemulihan. Contohnya *auto-restart*, *auto-scaling*, atau failover otomatis.
3. Redundansi sumber daya. Menyediakan cadangan server atau layanan.
4. Batasan dan isolasi kegagalan. Mencegah satu kegagalan menyebar ke seluruh sistem.
5. Evaluasi pasca-insiden. Setiap insiden digunakan sebagai bahan pembelajaran untuk peningkatan sistem.

Hubungan Manajemen Insiden, Resiliensi, dan SRE

Dalam pendekatan SRE, manajemen insiden dan resiliensi sistem berfungsi untuk:

- Menjaga layanan tetap berada dalam batas error budget,
- Memastikan SLO tetap tercapai,

- Mengurangi dampak gangguan terhadap pengguna.

Monitoring dan observability menjadi fondasi utama agar kedua proses ini dapat berjalan secara efektif.

Manajemen insiden memastikan gangguan ditangani secara sistematis, sementara resiliensi sistem memastikan layanan mampu bertahan dan pulih dari gangguan. Dengan dukungan monitoring, observability, dan prinsip SRE, sistem modern dapat beroperasi secara andal meskipun berada dalam lingkungan yang kompleks dan dinamis.

BAB IX
REKAYASA KINERJA DAN OPTIMASI SISTEM

Dalam sistem perangkat lunak modern yang melayani jutaan pengguna di atas infrastruktur terdistribusi, kinerja menjadi penentu utama pengalaman pengguna dan keberhasilan layanan, karena sistem yang lambat, boros sumber daya, atau tidak stabil di bawah beban tinggi akan kalah bersaing, sehingga performance engineering harus diintegrasikan sejak awal, bukan dijadikan tahap akhir. Bab ini menguraikan pendekatan sistematis untuk mengukur, menganalisis, dan mengoptimalkan kinerja, mulai dari dasar performance engineering, penggunaan stress testing dan load testing untuk memahami perilaku sistem, profiling dan benchmarking guna menemukan bottleneck, strategi optimasi kode dan infrastruktur, hingga isu efisiensi energi dan green computing sebagai respons terhadap meningkatnya konsumsi energi pusat data, dengan tujuan membekali pembaca untuk merancang sistem yang cepat, efisien, dan berkelanjutan.

A. Performance Engineering

Performance Engineering adalah pendekatan sistematis untuk merancang, mengukur, menganalisis, dan meningkatkan kinerja sistem perangkat lunak sepanjang seluruh siklus hidup pengembangan. Berbeda dengan optimasi kinerja yang bersifat reaktif dan dilakukan setelah masalah muncul, performance engineering bersifat proaktif. Kinerja dipertimbangkan sejak tahap perancangan arsitektur, pemilihan teknologi, hingga implementasi dan operasi sistem di lingkungan produksi.

Dalam sistem modern yang kompleks, kinerja tidak hanya berarti kecepatan respons, tetapi juga mencakup kemampuan sistem untuk mempertahankan performa yang stabil di bawah berbagai kondisi beban. Oleh karena itu, performance engineering menjadi disiplin penting dalam memastikan bahwa sistem mampu memenuhi ekspektasi pengguna dan kebutuhan bisnis secara konsisten.

Dimensi Kinerja Sistem

Kinerja sistem perangkat lunak memiliki beberapa dimensi utama yang saling berkaitan. Dimensi pertama adalah *latency*, yaitu waktu yang dibutuhkan sistem untuk merespons permintaan pengguna. Dimensi kedua adalah *throughput*, yang menggambarkan jumlah permintaan yang dapat diproses dalam satuan waktu tertentu. Selain itu, *resource utilization* menjadi dimensi penting untuk memahami seberapa efisien sistem menggunakan sumber daya seperti CPU, memori, dan jaringan.

Dimensi-dimensi ini harus dianalisis secara bersamaan. Peningkatan throughput, misalnya, dapat menyebabkan peningkatan latency jika tidak diimbangi dengan pengelolaan sumber daya yang baik. Performance engineering membantu menyeimbangkan dimensi-dimensi tersebut agar sistem tetap efisien dan responsif.

Kinerja sebagai Bagian dari Desain Sistem

Performance engineering menekankan bahwa keputusan arsitektur memiliki dampak langsung terhadap kinerja. Pemilihan arsitektur monolitik atau microservices, mekanisme komunikasi antar layanan, serta strategi penyimpanan data akan menentukan karakteristik kinerja

sistem. Oleh karena itu, kinerja harus menjadi pertimbangan sejak awal perancangan, bukan sekadar tahap optimasi akhir.

Pendekatan ini mendorong pengembang untuk memprediksi potensi bottleneck dan mengantisipasinya melalui desain yang tepat. Dengan demikian, sistem dapat tumbuh secara skalabel tanpa mengalami degradasi kinerja yang signifikan.

Pengukuran dan Evaluasi Kinerja

Pengukuran merupakan inti dari performance engineering. Tanpa data yang akurat, upaya optimasi akan bersifat spekulatif. Pengukuran kinerja dilakukan melalui berbagai teknik, seperti pengujian beban, pemantauan metrik sistem, dan analisis log. Data ini digunakan untuk mengevaluasi apakah sistem telah memenuhi target kinerja yang ditetapkan.

Evaluasi kinerja juga membantu organisasi memahami batas kemampuan sistem dan menentukan kapan perlu dilakukan peningkatan kapasitas atau perubahan arsitektur.

Peran Performance Engineering dalam Sistem Skala Besar

Dalam sistem skala besar, performance engineering berperan sebagai alat pengendali kompleksitas. Dengan pendekatan yang terstruktur, tim dapat mengelola pertumbuhan sistem tanpa mengorbankan pengalaman pengguna. Performance engineering memungkinkan organisasi untuk membuat keputusan berbasis data, mengoptimalkan penggunaan sumber daya, dan menjaga keseimbangan antara kinerja, biaya, dan keberlanjutan.

B. Stress Testing dan Load Testing

Stress testing dan load testing merupakan metode pengujian performa sistem yang bertujuan untuk memastikan bahwa aplikasi dan infrastruktur mampu menangani beban kerja sesuai kebutuhan serta tetap stabil ketika menghadapi kondisi ekstrem. Kedua pengujian ini penting untuk mengurangi risiko kegagalan sistem saat digunakan oleh banyak pengguna secara bersamaan.

Load Testing

Load testing adalah pengujian yang dilakukan dengan memberikan beban normal hingga beban maksimum yang direncanakan pada sistem. Tujuannya adalah untuk mengetahui apakah sistem mampu beroperasi dengan baik dalam kondisi penggunaan yang diharapkan.

Contoh load testing:

- Mensimulasikan 100, 500, atau 1.000 pengguna yang mengakses aplikasi secara bersamaan.
- Mengukur waktu respons (*response time*), tingkat error, dan penggunaan sumber daya sistem.

Dalam praktik, hasil load testing dipantau menggunakan metrik seperti:

- Latensi permintaan
- Penggunaan CPU dan memori
- System load
- Tingkat kegagalan permintaan

Jika seluruh metrik masih berada dalam batas Service Level Objective (SLO), maka sistem dianggap mampu menangani beban tersebut.

Stress Testing

Stress testing adalah pengujian yang dilakukan dengan memberikan beban melebihi kapasitas normal sistem, hingga sistem mengalami penurunan kinerja atau kegagalan. Tujuan utamanya bukan untuk menjaga sistem tetap berjalan, melainkan untuk memahami batas maksimum dan perilaku sistem saat gagal.

Contoh stress testing:

- Menjalankan simulasi ribuan permintaan dalam waktu singkat.
- Membebani CPU atau memori secara ekstrem.
- Mengisi disk hingga mendekati kapasitas maksimum.

Stress testing membantu menjawab pertanyaan seperti:

- Pada titik beban berapa sistem mulai melambat?
- Komponen mana yang gagal terlebih dahulu?
- Apakah sistem dapat pulih setelah beban dikurangi?

Perbedaan Load Testing dan Stress Testing

Aspek	**Load Testing**	**Stress Testing**
Purpose	Mengevaluasi performa sistem dalam beban yang diharapkan	Memeriksa perilaku sistem dalam beban ekstrem
Focus	Stabilitas, responsivitas, dan keandalan dalam penggunaan normal	Kerentanan, titik kegagalan, dan kemampuan pulih
Load Simulated	Lalu lintas pengguna dan	Beban berlebih atau

Aspek	Load Testing	Stress Testing
	skenario penggunaan yang wajar	abnormal yang melebihi kapasitas sistem
Goal	Memastikan sistem berjalan seperti yang diharapkan dalam permintaan normal	Menentukan bagaimana sistem gagal dan pulih di bawah tekanan
System Health	Menjaga sistem tetap berjalan dalam batas wajar	Mendorong sistem hingga ke batas atas atau melampauinya
Key Outcomes	Mendeteksi bottleneck performa dan memastikan stabilitas	Mendeteksi kerentanan tersembunyi dan kemampuan sistem menghadapi kegagalan
Resource Impact	Konsumsi resource ringan hingga sedang untuk simulasi realistis	Konsumsi resource tinggi untuk mensimulasikan kondisi ekstrem

Contoh Penerapan pada Sistem Nyata

Sebagai contoh, sebuah aplikasi web diuji menggunakan alat load testing. Selama pengujian:

- Prometheus mengumpulkan metrik CPU, memori, dan latensi.
- Grafana menampilkan peningkatan beban sistem secara visual.

Hasil pengujian menunjukkan bahwa:

- Sistem stabil hingga 500 pengguna bersamaan.
- Pada 800 pengguna, latensi meningkat signifikan.
- Pada 1.000 pengguna, sistem mengalami error dan restart.

Dari hasil ini, tim dapat menentukan kapasitas aman sistem dan melakukan perbaikan sebelum aplikasi digunakan secara luas.

Hubungan dengan Monitoring dan SRE

Stress testing dan load testing sangat berkaitan dengan:

- Monitoring, untuk mengamati dampak beban terhadap sistem.
- Observability, untuk memahami penyebab penurunan performa.
- SRE, untuk menentukan SLO, error budget, dan strategi peningkatan keandalan.

Data hasil pengujian sering digunakan sebagai dasar penetapan SLO yang realistis dan perencanaan kapasitas sistem.

Manfaat Stress Testing dan Load Testing

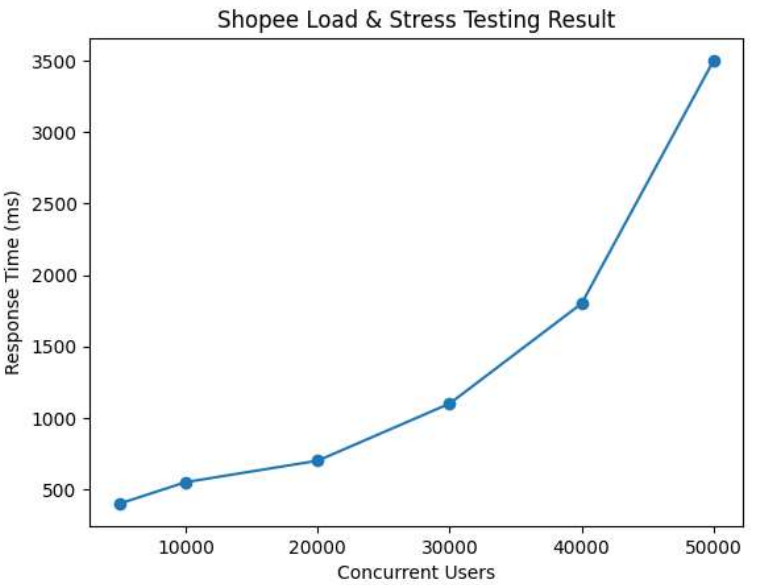
Manfaat utama dari kedua pengujian ini antara lain:

1. Mengetahui batas kemampuan sistem sebelum digunakan secara produksi.
2. Mengidentifikasi potensi bottleneck sejak dini.
3. Mengurangi risiko kegagalan saat lonjakan trafik.
4. Mendukung pengambilan keputusan teknis berbasis data.

Load testing memastikan sistem mampu menangani beban yang direncanakan, sedangkan stress testing menguji ketahanan sistem dalam kondisi ekstrem. Keduanya merupakan langkah penting dalam pengembangan sistem yang andal, terutama ketika dikombinasikan dengan monitoring dan prinsip SRE.

Contoh Skenario Pengujian Performa Shopee

1. Load & Stress Testing (Response Time vs Concurrent Users)



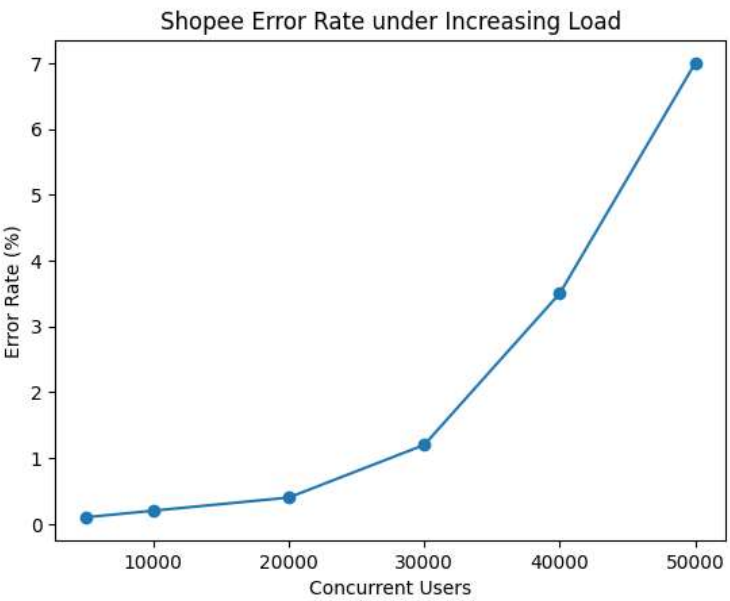
Kesimpulan teknis: Grafik menunjukkan bahwa hingga ±20.000 concurrent users, sistem Shopee masih berada pada zona stabil dengan waktu respons < 800 ms. Namun, setelah melewati 30.000 pengguna, terjadi kenaikan waktu respons yang tajam hingga >3 detik pada 50.000 pengguna. Ini menandakan titik jenuh (stress threshold) pada lapisan backend (database, search service, atau payment gateway).

Rekomendasi tindakan:

- Terapkan horizontal scaling pada service kritikal (checkout, payment, search).
- Optimalkan database sharding dan caching (Redis) untuk read-heavy traffic.

- Lakukan stress test lanjutan untuk menemukan breaking point per service, bukan hanya sistem secara keseluruhan.

2. Error Rate under Increasing Load



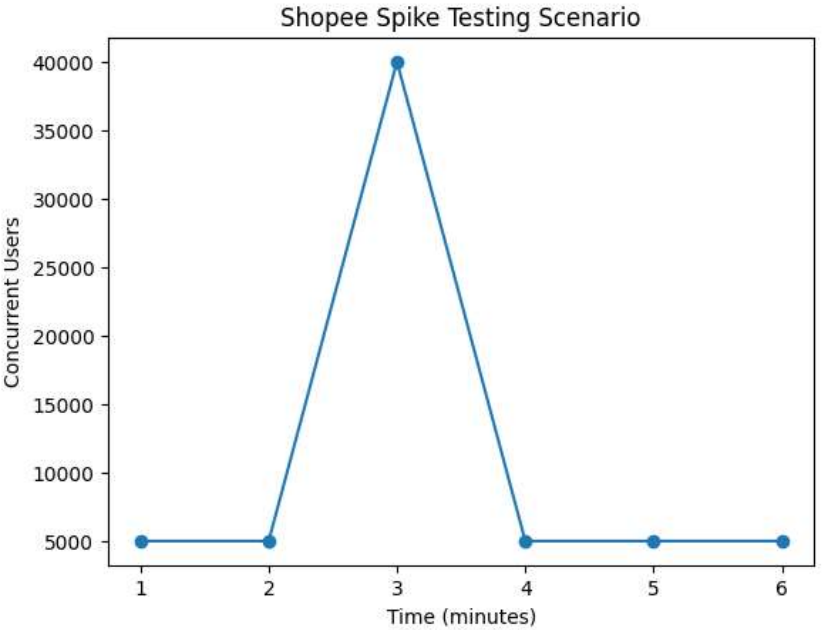
Kesimpulan teknis: Error rate meningkat eksponensial setelah 30.000 pengguna, mencapai 7% pada 50.000 pengguna. Ini mengindikasikan timeout, connection pool exhaustion, atau circuit breaker aktif. Pada konteks e-commerce, error rate >1% sudah berdampak langsung pada konversi transaksi.

Rekomendasi tindakan:

- Tambahkan rate limiting & graceful degradation (misalnya mematikan rekomendasi saat overload).

- Audit konfigurasi thread pool, DB connection pool, dan retry policy.
- Pisahkan jalur transaksi kritis (payment) dari non-kritis (review, rekomendasi).

3. Spike Testing (Lonjakan Trafik Mendadak)

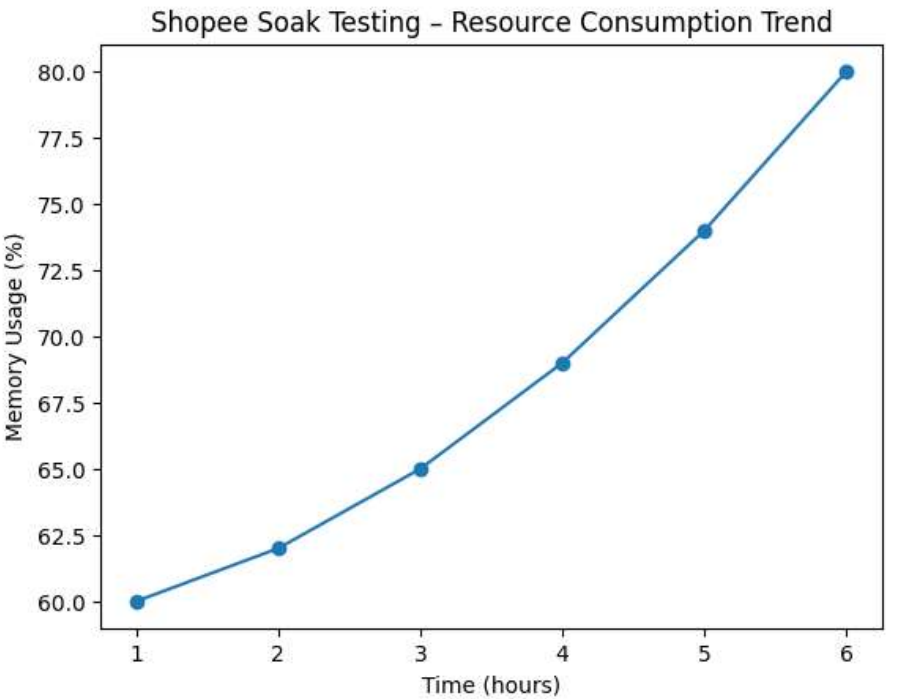


Kesimpulan teknis: Grafik spike menunjukkan lonjakan mendadak dari 5.000 ke 40.000 pengguna (flash sale scenario). Sistem mampu kembali stabil setelah spike, tetapi fase lonjakan berisiko menyebabkan cold start autoscaling, cache miss, dan request queue buildup.

Rekomendasi tindakan:

- Aktifkan predictive autoscaling sebelum event besar (11.11, 12.12).
- Pre-warm cache dan container/service penting.
- Terapkan queue-based load leveling untuk order & payment request.

4. Soak Testing (Beban Jangka Panjang)



Kesimpulan teknis: Grafik soak test menunjukkan kenaikan memori bertahap dari 60% ke 80% selama 6 jam. Pola ini

mengindikasikan potensi memory leak, object retention, atau cache eviction yang tidak optimal.

Rekomendasi tindakan:

- Lakukan heap dump & memory profiling pada service yang long-running.
- Evaluasi kebijakan cache TTL dan eviction.
- Jadwalkan rolling restart untuk service tertentu bila diperlukan.

Kesimpulan: Secara keseluruhan, hasil grafik menunjukkan bahwa arsitektur Shopee andal untuk beban normal, namun membutuhkan:

- optimasi skalabilitas pada traffic ekstrem,
- isolasi layanan kritis, dan
- kesiapan operasional berbasis data hasil testing.

Pendekatan ini memastikan performa tetap stabil, error terkendali, dan pengalaman pengguna tetap optimal saat event besar.

C. Profiling dan Benchmarking

Profiling dan benchmarking merupakan teknik analisis performa yang digunakan untuk memahami bagaimana sistem bekerja secara internal dan seberapa baik kinerjanya dibandingkan standar tertentu. Keduanya melengkapi stress testing dan load testing dengan memberikan informasi yang lebih mendalam dan terukur.

Profiling

Profiling adalah proses analisis untuk mengidentifikasi bagian sistem yang paling banyak menggunakan sumber daya, seperti CPU, memori, atau waktu eksekusi. Fokus utama profiling bukan pada

seberapa banyak beban yang diterima sistem, tetapi pada efisiensi komponen internal sistem.

Profiling umumnya digunakan untuk:

- Mengidentifikasi *bottleneck* pada kode atau layanan.
- Menemukan fungsi atau proses yang paling boros sumber daya.
- Mendeteksi masalah seperti *memory leak* atau pemrosesan berulang yang tidak efisien.

Contoh profiling:

- Analisis fungsi aplikasi yang menyebabkan lonjakan penggunaan CPU.
- Pemantauan alokasi memori selama aplikasi berjalan.
- Penelusuran jalur eksekusi yang memperlambat respons layanan.

Hasil profiling digunakan sebagai dasar optimasi sistem secara spesifik dan terarah.

Benchmarking

Benchmarking adalah proses pengujian performa sistem dengan metode dan skenario yang terstandar, kemudian membandingkan hasilnya dengan:

- Konfigurasi sistem sebelumnya,
- Sistem lain yang sejenis,
- Target performa yang telah ditetapkan.

Benchmarking membantu menjawab pertanyaan seperti:

- Apakah performa sistem meningkat setelah dilakukan optimasi?
- Seberapa besar pengaruh perubahan konfigurasi terhadap kinerja?
- Apakah sistem memenuhi standar performa yang diharapkan?

Contoh benchmarking:

- Mengukur waktu respons API sebelum dan sesudah peningkatan server.
- Membandingkan performa aplikasi pada dua jenis infrastruktur berbeda.
- Menguji performa database dengan konfigurasi indeks yang berbeda.

Contoh Penerapan pada Sistem Nyata

Sebagai contoh, sebuah aplikasi web menunjukkan latensi tinggi saat menerima banyak permintaan.

Langkah yang dilakukan:

1. Profiling digunakan untuk menemukan bahwa proses query database tertentu memakan waktu paling lama.
2. Optimasi dilakukan pada struktur query dan indeks database.
3. Benchmarking dilakukan untuk membandingkan waktu respons sebelum dan sesudah optimasi.
4. Hasil benchmarking menunjukkan penurunan latensi yang signifikan.

Pendekatan ini memastikan bahwa peningkatan performa benar-benar terukur dan bukan berdasarkan asumsi.

Hubungan dengan Monitoring dan Observability

Profiling dan benchmarking melengkapi sistem monitoring dan observability dengan:

- Memberikan penjelasan rinci tentang penyebab penurunan performa.
- Menyediakan data kuantitatif untuk evaluasi teknis.
- Mendukung pengambilan keputusan dalam konteks SRE dan perencanaan kapasitas.

Monitoring menunjukkan gejala masalah, sedangkan profiling dan benchmarking menjelaskan akar penyebab dan dampak perubahan.

D. Optimasi Kode dan Infrastruktur

1. Makna Optimasi dalam Sistem Perangkat Lunak

Optimasi kode dan infrastruktur merupakan upaya terencana untuk meningkatkan kinerja sistem tanpa mengubah fungsi utama aplikasi. Dalam praktiknya, banyak masalah kinerja tidak disebabkan oleh algoritma yang salah secara konseptual, tetapi oleh implementasi kode yang kurang efisien atau pemanfaatan infrastruktur yang tidak optimal. Oleh karena itu, optimasi harus dilakukan secara hati-hati dan berbasis data, bukan sekadar asumsi.

Optimasi yang baik berangkat dari pemahaman bahwa perangkat lunak dan infrastruktur adalah satu kesatuan. Kode yang efisien tetapi berjalan di atas infrastruktur yang salah konfigurasi tetap akan menghasilkan kinerja yang buruk, demikian pula sebaliknya.

2. Optimasi pada Level Kode

Optimasi kode berfokus pada bagaimana instruksi dijalankan oleh sistem. Praktik optimasi yang umum meliputi:

- penyederhanaan algoritma dan struktur data,
- pengurangan operasi yang berulang atau tidak perlu,
- pengelolaan memori yang lebih efisien,
- pengurangan *blocking operations* dalam proses paralel atau asinkron.

Optimasi kode tidak selalu berarti menulis kode yang kompleks. Justru, kode yang sederhana, jelas, dan terstruktur sering kali lebih mudah dioptimalkan dan dipelihara. Prinsip penting dalam optimasi kode adalah mengoptimalkan bagian yang benar-benar menjadi bottleneck, bukan seluruh sistem secara membabi buta.

3. Optimasi Akses Data dan I/O

Dalam banyak sistem modern, bottleneck kinerja sering muncul pada akses data dan operasi input–output. Optimasi pada area ini mencakup:

- pengurangan jumlah akses ke basis data,
- penggunaan mekanisme *batch processing*,
- penerapan caching pada data yang sering diakses,
- pemilihan format data dan protokol komunikasi yang lebih efisien.

Dengan mengoptimalkan alur data, sistem dapat mengurangi latensi secara signifikan tanpa harus meningkatkan kapasitas perangkat keras.

4. Optimasi Infrastruktur

Optimasi infrastruktur berfokus pada pemanfaatan sumber daya komputasi secara efisien. Hal ini mencakup pengaturan kapasitas server, konfigurasi jaringan, serta pemilihan layanan *cloud* yang sesuai dengan karakteristik beban kerja. Strategi umum meliputi:

- penskalaan sumber daya secara horizontal atau vertikal,
- pemanfaatan *load balancing* untuk distribusi beban,
- penggunaan container dan orkestrasi untuk efisiensi deployment,
- penyesuaian konfigurasi sistem operasi dan runtime.

Optimasi infrastruktur memungkinkan sistem mencapai kinerja optimal tanpa pemborosan sumber daya, sehingga berdampak langsung pada efisiensi biaya operasional.

5. Sinergi Optimasi Kode dan Infrastruktur

Optimasi yang efektif menuntut sinergi antara kode dan infrastruktur. Perubahan pada satu sisi sering kali memengaruhi sisi lainnya. Misalnya, optimasi kode yang mengurangi konsumsi memori dapat memungkinkan pengurangan ukuran instans server, sedangkan optimasi infrastruktur dapat membuka peluang untuk paralelisme kode yang lebih baik.

Pendekatan terpadu ini membantu organisasi mencapai keseimbangan antara kinerja, biaya, dan skalabilitas, serta memastikan bahwa optimasi tidak bersifat sementara, tetapi berkelanjutan seiring pertumbuhan sistem.

6. Posisi Optimasi dalam Siklus Hidup Sistem

Optimasi kode dan infrastruktur bukan aktivitas satu kali, melainkan proses berkelanjutan. Setiap perubahan fitur, pertumbuhan pengguna, atau evolusi arsitektur dapat memunculkan kebutuhan optimasi baru. Oleh karena itu, optimasi harus dipandang sebagai bagian integral dari rekayasa kinerja jangka panjang, yang akan dilengkapi oleh pembahasan selanjutnya mengenai optimasi konsumsi energi dan *green computing*.

E. Optimasi Konsumsi Energi

Seiring pertumbuhan sistem perangkat lunak berskala besar dan pemanfaatan *cloud computing*, konsumsi energi menjadi isu penting yang tidak dapat diabaikan. Pusat data modern mengoperasikan ribuan server yang berjalan secara terus-menerus, sehingga setiap peningkatan beban komputasi berdampak langsung pada konsumsi listrik dan emisi karbon. Oleh karena itu, optimasi konsumsi energi tidak hanya berkaitan dengan efisiensi biaya operasional, tetapi juga dengan tanggung jawab lingkungan dan keberlanjutan sistem digital.

Optimasi konsumsi energi dalam konteks rekayasa perangkat lunak berfokus pada bagaimana desain, implementasi, dan pengoperasian sistem dapat mengurangi penggunaan energi tanpa mengorbankan kualitas layanan.

Hubungan antara Kinerja dan Konsumsi Energi

Kinerja dan konsumsi energi memiliki hubungan yang erat. Sistem yang tidak efisien secara komputasi cenderung menggunakan lebih banyak CPU, memori, dan operasi I/O, yang pada akhirnya meningkatkan konsumsi daya. Sebaliknya, sistem yang dirancang

dengan baik dapat menyelesaikan pekerjaan lebih cepat dan kembali ke kondisi idle, sehingga total energi yang digunakan menjadi lebih rendah.

Namun, optimasi kinerja tidak selalu identik dengan penghematan energi. Peningkatan kinerja melalui penambahan sumber daya berlebih justru dapat meningkatkan konsumsi energi. Oleh karena itu, optimasi energi menuntut keseimbangan antara kecepatan, kapasitas, dan efisiensi penggunaan sumber daya.

Strategi Optimasi Energi pada Level Perangkat Lunak

Pada level perangkat lunak, optimasi konsumsi energi dapat dilakukan melalui:

- penulisan kode yang lebih efisien dan minim komputasi berulang,
- pengurangan proses latar belakang yang tidak perlu,
- pengelolaan *idle state* aplikasi secara optimal,
- penjadwalan tugas komputasi secara cerdas.

Aplikasi yang mampu mengatur kapan harus aktif dan kapan harus berada pada mode hemat energi akan memberikan kontribusi signifikan terhadap pengurangan konsumsi daya secara keseluruhan.

Optimasi Energi melalui Infrastruktur dan Operasi

Selain kode, optimasi energi juga sangat dipengaruhi oleh pengelolaan infrastruktur. Praktik umum meliputi:

- konsolidasi beban kerja untuk mengurangi jumlah server aktif,
- pemanfaatan mekanisme *auto-scaling* untuk menyesuaikan kapasitas dengan kebutuhan aktual,

- penggunaan *virtualization* dan *containerization* untuk meningkatkan densitas pemanfaatan server,
- pemilihan lokasi pusat data yang memiliki efisiensi energi tinggi.

Pendekatan ini memungkinkan sistem beroperasi secara adaptif, hanya menggunakan energi sebesar yang benar-benar dibutuhkan.

Peran Pengukuran dan Monitoring Energi

Optimasi energi yang efektif membutuhkan pengukuran yang akurat. Monitoring konsumsi energi memungkinkan tim untuk:

- mengidentifikasi komponen sistem yang boros energi,
- mengevaluasi dampak perubahan kode atau konfigurasi,
- membuat keputusan berbasis data terkait efisiensi sistem.

Tanpa visibilitas terhadap konsumsi energi, upaya optimasi berisiko tidak tepat sasaran dan sulit dievaluasi dampaknya.

Optimasi Energi sebagai Bagian dari Keberlanjutan Sistem

Optimasi konsumsi energi tidak dapat dipisahkan dari tujuan keberlanjutan jangka panjang. Dengan menurunkan kebutuhan energi, organisasi dapat mengurangi biaya operasional sekaligus dampak lingkungan dari sistem digital yang dijalankan.

F. Green Computing dalam Sistem Skala Besar

Dalam konteks sistem perangkat lunak skala besar, *green computing* tidak merujuk langsung pada perangkat keras hemat energi, melainkan pada bagaimana perangkat lunak dirancang dan dijalankan agar meminimalkan konsumsi energi dari infrastruktur yang

digunakannya. Perangkat lunak berperan sebagai pengendali utama beban komputasi: ia menentukan kapan prosesor bekerja, berapa lama memori digunakan, seberapa sering data dipindahkan, dan berapa banyak sumber daya yang tetap aktif.

Dengan demikian, green computing pada level perangkat lunak berarti menghasilkan fungsi dan nilai yang sama dengan penggunaan sumber daya komputasi yang lebih rendah, sehingga secara tidak langsung menurunkan konsumsi energi dan jejak karbon sistem.

Sistem skala besar seperti layanan *cloud*, platform digital global, dan pusat data modern beroperasi secara kontinu dan melayani jutaan permintaan. Pada skala ini, inefisiensi kecil dalam desain perangkat lunak dapat berakumulasi menjadi konsumsi energi yang sangat besar. Sebuah keputusan arsitektural yang kurang tepat dapat mengunci sistem pada pola pemborosan energi selama bertahun-tahun.

Green Computing melalui Desain dan Arsitektur Sistem

Green computing dimulai dari desain arsitektur. Sistem yang dirancang untuk dapat menskalakan sumber daya secara dinamis, mematikan komponen yang tidak aktif, dan mengisolasi beban kerja akan menggunakan energi secara lebih efisien. Arsitektur berbasis layanan, pemrosesan berbasis peristiwa, dan desain yang mendukung *on-demand execution* merupakan contoh pendekatan yang mendukung prinsip green computing.

Sebaliknya, arsitektur yang memaksa seluruh sistem tetap aktif meskipun beban rendah cenderung menghasilkan konsumsi energi yang konstan dan tidak efisien.

Peran Perangkat Lunak dalam Efisiensi Infrastruktur

Pada sistem skala besar, perangkat lunak menentukan bagaimana infrastruktur digunakan. Mekanisme seperti *autoscaling*, penjadwalan beban kerja, serta pengelolaan *idle state* dikendalikan sepenuhnya oleh perangkat lunak. Dengan pengaturan yang tepat, sistem dapat mengonsolidasikan beban kerja pada sejumlah kecil sumber daya dan membebaskan sisanya untuk tidak aktif.

Pendekatan ini memungkinkan pengurangan energi tanpa harus menurunkan kualitas layanan, karena sistem tetap dapat merespons peningkatan beban secara adaptif.

Green computing mendorong perubahan cara pandang dalam pengambilan keputusan teknis. Evaluasi teknologi tidak lagi hanya berfokus pada kinerja dan biaya, tetapi juga pada efisiensi energi jangka panjang. Pilihan algoritma, pola komunikasi antar layanan, serta strategi penyimpanan dan pemrosesan data memiliki implikasi langsung terhadap konsumsi energi sistem.

Dengan memasukkan dimensi energi ke dalam evaluasi teknis, organisasi dapat membangun sistem yang lebih bertanggung jawab dan berkelanjutan.

Dalam jangka panjang, green computing merupakan bagian dari strategi keberlanjutan digital. Sistem perangkat lunak skala besar yang dirancang dengan prinsip efisiensi energi akan lebih siap menghadapi tekanan biaya, regulasi lingkungan, dan tuntutan masyarakat terhadap praktik teknologi yang berkelanjutan. Dengan demikian, green computing melengkapi rekayasa kinerja dan optimasi sistem sebagai pendekatan holistik untuk membangun sistem yang efisien, andal, dan berorientasi masa depan.

BAB X
ARSITEKTUR BERBASIS KECERDASAN BUATAN (AI) DAN OTOMASI

Perkembangan kecerdasan buatan (Artificial Intelligence/AI) telah mentransformasi perangkat lunak dari sistem pasif menjadi sistem yang mampu belajar, beradaptasi, dan mengambil keputusan secara otomatis. Dalam rekayasa perangkat lunak modern, AI tidak lagi diposisikan sebagai fitur tambahan, melainkan sebagai bagian integral dari arsitektur sistem yang memengaruhi alur utama pemrosesan. Integrasi AI menuntut pendekatan arsitektural yang lebih dinamis karena melibatkan data berskala besar, model pembelajaran, dan proses inferensi yang terus berkembang. Bab ini membahas peran AI dalam rekayasa perangkat lunak, konsep *intelligent system architecture*, MLOps, otomasi pengujian, *self-healing system*, serta tantangan etika dan keamanan, guna menunjukkan bagaimana AI membentuk ulang desain dan operasi arsitektur perangkat lunak modern.

A. Peran AI dalam Rekayasa Perangkat Lunak

1. Perubahan Paradigma Rekayasa Perangkat Lunak

Kehadiran kecerdasan buatan menggeser rekayasa perangkat lunak dari pendekatan *rule-based* menuju *data-driven*. Perilaku sistem tidak lagi sepenuhnya ditentukan oleh kode, tetapi dipelajari dari data melalui model. Dampaknya, rekayasa perangkat lunak kini mencakup pengelolaan data, validitas model, serta mekanisme pembelajaran berkelanjutan, tidak hanya penulisan dan pemeliharaan kode.

2. AI sebagai Komponen Arsitektural

Dalam sistem modern, AI berperan sebagai komponen inti arsitektur yang memengaruhi alur utama sistem. Model AI berfungsi sebagai pengambil keputusan atau prediktor, sehingga arsitektur harus mendukung siklus hidup model, latensi inferensi, skalabilitas, dan integrasi dengan layanan lain.

3. Peran AI dalam Otomasi Proses Pengembangan

AI mendukung otomasi berbagai aktivitas pengembangan seperti analisis kode, pengujian, dan deteksi kesalahan. Otomasi ini meningkatkan produktivitas dan mengurangi kesalahan manusia, sekaligus mempercepat siklus rilis tanpa menurunkan kualitas perangkat lunak.

4. AI dalam Operasi dan Pemeliharaan Sistem

Pada tahap operasional, AI digunakan untuk mendeteksi anomali, memprediksi kegagalan, dan mengoptimalkan sumber daya. Pendekatan ini memungkinkan sistem beradaptasi secara dinamis dan menjadi dasar pengembangan *self-healing systems*.

5. Tantangan Rekayasa akibat Integrasi AI

Integrasi AI menghadirkan tantangan baru seperti ketidakpastian perilaku model, ketergantungan pada data, dan kebutuhan evaluasi berkelanjutan. Oleh karena itu, metodologi dan arsitektur rekayasa perangkat lunak perlu berevolusi agar sistem berbasis AI tetap dapat dipahami, diuji, dikendalikan, serta bertanggung jawab.

B. Intelligent System Architecture

Intelligent System Architecture adalah pendekatan perancangan sistem perangkat lunak yang memungkinkan sistem mengamati, memahami, dan merespons lingkungan secara adaptif menggunakan kecerdasan buatan. Berbeda dengan arsitektur tradisional yang perilakunya ditentukan oleh aturan statis, arsitektur cerdas mengintegrasikan data, model AI, dan mekanisme pengambilan keputusan sebagai bagian inti dari struktur sistem.

Intinya, sistem tidak hanya "menjalankan perintah", tetapi belajar dari data dan menyesuaikan tindakannya seiring waktu.

Komponen Utama dalam Intelligent System Architecture

Arsitektur sistem cerdas umumnya terdiri dari beberapa komponen kunci:

- **Data layer**: sumber data operasional dan historis yang menjadi bahan pembelajaran.
- **Model layer**: model AI/ML yang melakukan prediksi, klasifikasi, atau rekomendasi.
- **Decision layer**: logika yang mengubah output model menjadi tindakan sistem.
- **Execution layer**: layanan atau komponen yang mengeksekusi keputusan.
- **Feedback loop**: mekanisme umpan balik untuk memperbaiki model dan keputusan.

Kehadiran *feedback loop* membedakan sistem cerdas dari sistem konvensional karena memungkinkan pembelajaran berkelanjutan.

Pola Interaksi Data–Model–Layanan

Dalam arsitektur cerdas, alur kerja tidak linier. Data mengalir dari sistem operasional ke model, hasil inferensi memengaruhi keputusan layanan, dan hasil eksekusi kembali menghasilkan data baru. Pola ini membentuk siklus adaptif yang terus berulang.

Pendekatan ini menuntut desain yang memisahkan model dari layanan bisnis (decoupling), sehingga model dapat diperbarui tanpa mengganggu operasi sistem secara keseluruhan.

Skalabilitas dan Kinerja Sistem Cerdas

Sistem cerdas sering kali harus melayani inferensi real-time dan pemrosesan data berskala besar. Oleh karena itu, arsitektur harus mempertimbangkan:

- pemisahan beban pelatihan dan inferensi,
- penskalaan komponen model secara independen,
- pengelolaan latensi agar keputusan tetap cepat dan relevan.

Keputusan arsitektural ini memastikan bahwa kecerdasan sistem tidak mengorbankan kinerja dan keandalan.

Integrasi dengan Sistem Non-AI

Tidak semua bagian sistem harus cerdas. Intelligent System Architecture dirancang untuk berkoeksistensi dengan komponen non-AI, seperti aturan bisnis deterministik dan layanan transaksional. Integrasi ini menjaga stabilitas sistem dan memungkinkan penggunaan AI secara selektif pada area yang benar-benar membutuhkan adaptasi dan prediksi. Pendekatan hibrida ini membantu organisasi mengadopsi AI secara bertahap dan terkontrol.

Merancang arsitektur sistem cerdas menghadirkan tantangan khusus, seperti ketergantungan pada kualitas data, perubahan perilaku model dari waktu ke waktu, serta kebutuhan akan observability dan kontrol yang lebih kuat. Arsitektur harus menyediakan mekanisme untuk memantau kinerja model dan mencegah dampak negatif dari keputusan otomatis yang tidak diinginkan.

C. MLOps dan Pipeline Model

MLOps (Machine Learning Operations) merupakan pendekatan yang mengadaptasi prinsip DevOps untuk pengembangan dan operasional sistem berbasis pembelajaran mesin. Tujuan utama MLOps adalah memastikan bahwa model kecerdasan buatan dapat dikembangkan, diuji, diterapkan, dan dipelihara secara andal dalam lingkungan produksi. Berbeda dengan perangkat lunak konvensional, sistem berbasis AI tidak hanya bergantung pada kode, tetapi juga pada data dan model yang dapat berubah seiring waktu.

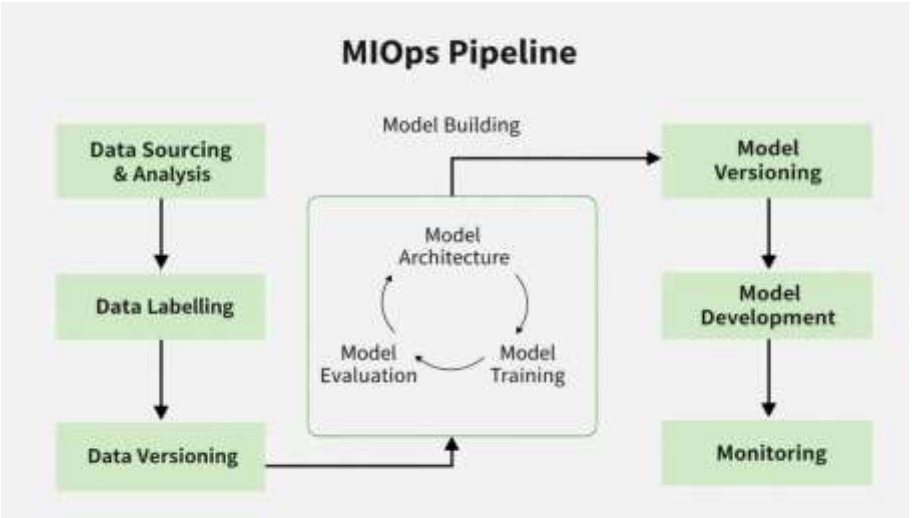
Model AI yang telah dilatih dengan baik belum tentu siap digunakan di produksi. Beberapa tantangan utama meliputi:

- Perubahan karakteristik data (*data drift*).
- Penurunan akurasi model (*model degradation*).
- Kesulitan reproduksi hasil pelatihan.
- Kurangnya visibilitas terhadap performa model di produksi.

MLOps hadir untuk mengatasi tantangan tersebut melalui proses yang terstandar dan terotomasi.

Pipeline Model dalam MLOps

Pipeline model adalah rangkaian proses terstruktur yang mengelola siklus hidup model AI secara end-to-end. Pipeline ini umumnya mencakup tahapan berikut:



Gambar 18. Pipeline dalam MLOps

1. **Pengumpulan dan Validasi Data**. Data dikumpulkan dari berbagai sumber dan divalidasi untuk memastikan kualitas dan konsistensinya.

2. **Pelatihan Model**. Model dilatih menggunakan data yang telah diproses dengan konfigurasi yang terdokumentasi.

3. **Evaluasi Model**. Kinerja model diukur menggunakan metrik yang relevan, seperti akurasi, presisi, atau latensi.

4. **Deployment Model**. Model yang lolos evaluasi diterapkan ke lingkungan produksi secara terkontrol.
5. **Monitoring Model**. Performa model dipantau secara berkelanjutan untuk mendeteksi *drift* atau anomali.
6. **Pembaruan dan Retraining**. Model diperbarui dan dilatih ulang ketika performa menurun atau data berubah signifikan.

Contoh Penerapan MLOps

Sebagai contoh, sebuah sistem rekomendasi menggunakan model pembelajaran mesin untuk menyarankan konten kepada pengguna.

Dalam praktik MLOps:

- Data interaksi pengguna dikumpulkan secara berkala.
- Model dilatih ulang setiap periode tertentu.
- Performa model dimonitor melalui metrik akurasi dan waktu respons.
- Model lama digantikan secara bertahap jika model baru menunjukkan kinerja lebih baik.

Pendekatan ini memastikan sistem tetap relevan dan responsif terhadap perubahan perilaku pengguna.

Hubungan MLOps dengan Monitoring dan SRE

MLOps sangat bergantung pada monitoring dan observability untuk:

- Menilai stabilitas layanan berbasis AI.
- Mengelola risiko kegagalan model.

- Menjaga sistem tetap dalam batas Service Level Objective (SLO).

Dalam konteks SRE, model AI diperlakukan sebagai komponen layanan yang juga memiliki target keandalan dan performa.

D. Otomasi Pengujian Berbasis AI

Pengujian perangkat lunak pada sistem modern yang kompleks dan cepat berubah semakin sulit ditangani dengan pendekatan tradisional berbasis skrip statis. Dalam konteks ini, AI dimanfaatkan untuk mengotomatisasi dan meningkatkan efektivitas pengujian dengan membuat prosesnya lebih adaptif. Berbeda dari pengujian konvensional yang kaku dan reaktif terhadap perubahan, pengujian berbasis AI mampu mempelajari pola perilaku aplikasi sehingga lebih toleran terhadap perubahan antarmuka dan alur sistem.

AI berperan dalam pembuatan dan pemeliharaan *test case*. Berdasarkan analisis kode, log, dan data penggunaan, AI dapat menghasilkan serta memprioritaskan *test case* berisiko tinggi, sekaligus menyesuaikan skrip yang rusak akibat perubahan sistem. Selain itu, AI efektif dalam mendeteksi cacat dan anomali yang sulit ditemukan secara manual melalui analisis data historis dan pola kegagalan.

Dalam praktiknya, otomasi pengujian berbasis AI diintegrasikan ke dalam pipeline CI/CD. AI membantu menentukan cakupan dan jenis pengujian yang relevan pada setiap perubahan kode, sehingga siklus rilis dapat dipercepat tanpa menurunkan kualitas. Pendekatan ini selaras dengan prinsip DevOps dan MLOps dalam pengembangan perangkat lunak modern.

E. Self-Healing System

Self-healing system adalah sistem yang mampu mendeteksi gangguan, menganalisis penyebabnya, dan melakukan pemulihan secara otomatis tanpa atau dengan intervensi manusia yang minimal. Konsep ini bertujuan untuk meningkatkan keandalan layanan, mengurangi waktu henti (*downtime*), dan mempercepat respons terhadap insiden.

Dalam arsitektur modern, self-healing system menjadi komponen penting untuk menghadapi kompleksitas sistem terdistribusi dan microservices.

Prinsip Dasar Self-Healing System

Self-healing system dibangun berdasarkan beberapa prinsip utama:

1. Deteksi otomatis terhadap kegagalan atau anomali.
2. Keputusan berbasis aturan atau AI untuk menentukan tindakan pemulihan.
3. Eksekusi pemulihan otomatis, seperti restart atau failover.
4. Evaluasi pasca-pemulihan untuk perbaikan berkelanjutan.

Sistem tidak diasumsikan selalu berjalan sempurna, tetapi dirancang untuk pulih dengan cepat saat kegagalan terjadi.

Peran Monitoring dan Observability

Self-healing sangat bergantung pada monitoring dan observability. Data metrik, log, dan event digunakan untuk:

- Mendeteksi kondisi tidak normal.

- Mengidentifikasi pola kegagalan berulang.
- Menentukan ambang batas pemulihan (*threshold*).

Sebagai contoh, lonjakan penggunaan memori yang terus meningkat dapat dikenali sebagai indikasi potensi kegagalan sebelum sistem berhenti berfungsi.

Mekanisme Pemulihan Otomatis (Contoh Konkret)

Contoh penerapan self-healing pada sistem nyata:

1. **Deteksi**. Monitoring mendeteksi bahwa penggunaan memori mendekati batas kritis.
2. **Analisis**. Sistem mengidentifikasi bahwa layanan tertentu mengalami kebocoran memori.
3. **Tindakan Pemulihan**. Layanan direstart secara otomatis atau dialihkan ke instans cadangan.
4. **Verifikasi**. Sistem memastikan layanan kembali ke kondisi normal.
5. **Pencatatan dan Pembelajaran**. Insiden dicatat untuk evaluasi dan peningkatan sistem.

Pendekatan ini mengurangi kebutuhan respons manual dan mempercepat pemulihan layanan.

Self-Healing dan SRE

Dalam pendekatan SRE, self-healing system berfungsi untuk:

- Menjaga sistem tetap berada dalam batas error budget.
- Mengurangi frekuensi dan dampak insiden.
- Memungkinkan tim fokus pada peningkatan sistem, bukan pemadaman kebakaran operasional.

BAB XI
MANAJEMEN PROYEK PERANGKAT LUNAK SKALA BESAR

Pengembangan perangkat lunak skala besar tidak hanya bergantung pada kecanggihan teknologi dan arsitektur, tetapi sangat ditentukan oleh efektivitas manajemen proyek. Proyek enterprise umumnya melibatkan banyak tim, pemangku kepentingan yang beragam, kebutuhan bisnis yang terus berubah, serta risiko teknis dan organisasional yang kompleks. Kegagalan sering kali bersumber dari lemahnya perencanaan, koordinasi, pengambilan keputusan, dan tata kelola, bukan dari keterbatasan teknologi. Bab ini membahas prinsip dan praktik manajemen proyek perangkat lunak skala besar, meliputi penerapan Agile di organisasi besar, peran DevOps dalam kolaborasi lintas tim, manajemen risiko, estimasi biaya dan waktu, governance TI, serta kematangan proses sebagai indikator keberlanjutan pengelolaan proyek kompleks.

A. Agile pada Proyek Enterprise

Konteks Agile dalam Skala Enterprise

Agile yang awalnya ditujukan untuk tim kecil kini diadopsi pada proyek enterprise untuk merespons kompleksitas dan perubahan bisnis yang cepat. Pada skala ini, Agile berfungsi sebagai kerangka manajemen proyek yang mengoordinasikan banyak tim, produk, dan pemangku kepentingan, sehingga menuntut penyesuaian struktur organisasi, proses keputusan, dan budaya kolaboratif.

Nilai Agile dan Relevansinya untuk Enterprise

Nilai Agile kolaborasi, adaptasi terhadap perubahan, dan penyampaian nilai bertahap tetap relevan. Implementasinya perlu diseimbangkan dengan perencanaan jangka panjang, kepatuhan regulasi, dan koordinasi lintas unit. Pendekatan iteratif membantu memecah proyek kompleks untuk evaluasi berkala dan mitigasi risiko.

Tantangan Implementasi Agile Skala Besar

Tantangan utama meliputi koordinasi lintas tim (sering tersebar geografis), konflik antara fleksibilitas Agile dan kebutuhan dokumentasi, serta menjaga konsistensi visi produk. Tanpa sinkronisasi yang baik, Agile berisiko menimbulkan fragmentasi hasil.

Pendekatan Agile Skala Enterprise

Pendekatan Agile skala besar menekankan struktur koordinasi, kepemimpinan produk yang jelas, dan komunikasi terstandar. Prinsip Agile dipertahankan, namun diperluas untuk mengelola kompleksitas organisasi besar dan menyeimbangkan fleksibilitas dengan kontrol.

Peran Manajemen dalam Agile Enterprise

Manajemen beralih dari pengendali tugas menjadi fasilitator strategis: menyelaraskan tujuan bisnis, menghilangkan hambatan lintas tim, dan memastikan ketersediaan sumber daya. Keberhasilan bergantung pada dukungan kepemimpinan dan budaya yang adaptif.

Dampak Agile terhadap Keberhasilan Proyek Skala Besar

Penerapan yang tepat meningkatkan transparansi, mempercepat nilai, dan menurunkan risiko kegagalan. Agile membantu enterprise beradaptasi tanpa kehilangan kendali strategis, menjadi landasan menuju integrasi DevOps dan kolaborasi tim yang lebih kuat.

B. DevOps dan Kolaborasi Tim

Dalam proyek perangkat lunak skala besar, pemisahan yang kaku antara tim pengembangan (*development*) dan tim operasi (*operations*) sering kali menjadi sumber keterlambatan, kesalahan, dan konflik. DevOps muncul sebagai pendekatan untuk menyatukan pengembangan dan operasi dalam satu alur kerja yang kolaboratif. Tujuan utamanya adalah mempercepat penyampaian perangkat lunak tanpa mengorbankan stabilitas dan keandalan sistem.

Pada skala enterprise, DevOps bukan sekadar kumpulan alat, melainkan budaya kerja yang menekankan kolaborasi lintas fungsi, tanggung jawab bersama, dan perbaikan berkelanjutan.

DevOps sebagai Pendekatan Kolaboratif

DevOps mendorong kolaborasi yang erat antara pengembang, tim operasi, penguji, dan pihak lain yang terlibat dalam siklus hidup perangkat lunak. Dengan pendekatan ini, masalah operasional dipertimbangkan sejak tahap pengembangan, sehingga mengurangi risiko kegagalan di lingkungan produksi.

Kolaborasi ini juga meningkatkan pemahaman bersama tentang sistem secara keseluruhan, yang sangat penting dalam proyek skala besar dengan banyak komponen dan dependensi.

Peran Automasi dalam DevOps

Automasi merupakan pilar utama DevOps. Proses-proses seperti *build*, *testing*, *deployment*, dan pemantauan diotomatisasi untuk mengurangi kesalahan manual dan mempercepat siklus rilis. Dalam proyek enterprise, automasi membantu menjaga konsistensi proses meskipun jumlah tim dan frekuensi rilis meningkat. Dengan automasi, tim dapat lebih fokus pada pengembangan nilai bisnis dan peningkatan kualitas sistem.

C. Manajemen Risiko Sistem Skala Besar

Manajemen risiko merupakan elemen krusial dalam proyek perangkat lunak skala besar karena sistem jenis ini melibatkan banyak komponen, tim lintas fungsi, serta ketergantungan teknis dan organisasional yang kompleks. Risiko tidak hanya bersumber dari aspek teknologi, tetapi juga dari ketidaksinkronan tujuan bisnis, keterbatasan sumber daya, dinamika organisasi, serta perubahan lingkungan eksternal. Pada sistem skala besar, risiko bersifat saling terkait, sehingga kegagalan pada satu komponen dapat memicu dampak berantai yang memengaruhi keseluruhan sistem.

Risiko dalam proyek perangkat lunak skala besar umumnya dapat diklasifikasikan menjadi risiko teknis, manajerial, operasional, dan bisnis. Risiko teknis mencakup kompleksitas arsitektur, masalah skalabilitas, integrasi, dan kinerja. Risiko manajerial berkaitan dengan perencanaan yang tidak realistis, koordinasi tim yang lemah, serta perubahan ruang lingkup yang tidak terkendali. Risiko operasional meliputi gangguan layanan, ketergantungan pihak ketiga, dan keamanan, sementara risiko bisnis muncul dari perubahan kebutuhan pengguna, tekanan pasar, dan regulasi. Identifikasi risiko sejak awal

secara kolaboratif menjadi kunci untuk melihat potensi masalah dari berbagai perspektif.

Setelah diidentifikasi, risiko perlu dianalisis dan diprioritaskan berdasarkan tingkat dampak dan probabilitasnya. Strategi mitigasi kemudian dirancang, seperti penyempurnaan arsitektur untuk mengurangi ketergantungan, penerapan pengujian dan automasi, penguatan koordinasi lintas tim, serta penyusunan rencana kontinjensi. Manajemen risiko harus dijalankan secara berkelanjutan seiring evolusi proyek, sehingga sistem tetap terkendali, adaptif, dan memiliki peluang keberhasilan yang lebih tinggi dalam jangka panjang.

D. Estimasi Biaya dan Waktu

Estimasi biaya dan waktu merupakan tantangan krusial dalam proyek perangkat lunak skala besar karena tingginya kompleksitas sistem, ketergantungan antar tim, serta ketidakpastian kebutuhan. Ketidakakuratan estimasi sering memicu keterlambatan, pembengkakan anggaran, dan penurunan kualitas. Oleh sebab itu, estimasi tidak dapat diperlakukan sebagai aktivitas sekali jadi, melainkan sebagai proses dinamis yang perlu diperbarui seiring perkembangan proyek dan perubahan konteks teknis maupun bisnis.

Pada proyek skala besar, estimasi dipengaruhi oleh faktor teknis dan non-teknis, seperti kompleksitas arsitektur, ukuran dan keahlian tim, teknologi yang digunakan, serta koordinasi lintas organisasi dan perubahan prioritas bisnis. Pendekatan estimasi umumnya mengombinasikan pengalaman historis, model kuantitatif berbasis ukuran sistem, dan estimasi iteratif ala Agile yang memungkinkan penyesuaian bertahap berdasarkan capaian aktual. Kombinasi ini membantu mengurangi bias dan meningkatkan realisme perencanaan.

Estimasi yang efektif dilakukan secara bertahap dan transparan dengan membagi proyek ke modul yang lebih kecil, melibatkan pemangku kepentingan teknis dan manajerial, menyediakan cadangan waktu dan biaya, serta melakukan revisi berkala. Selain sebagai alat perencanaan, estimasi berfungsi sebagai sarana komunikasi untuk mengelola ekspektasi dan mendukung pengambilan keputusan strategis. Pendekatan adaptif dan berbasis data meningkatkan peluang proyek diselesaikan tepat waktu, sesuai anggaran, dan dengan kualitas yang terjaga.

E. Governance TI

Governance TI merupakan kerangka pengelolaan yang memastikan pengembangan dan operasional perangkat lunak skala besar selaras dengan tujuan strategis organisasi. Pada proyek berskala besar, keputusan teknis berdampak langsung pada risiko bisnis, kepatuhan, keamanan, dan keberlanjutan sistem. Karena itu, governance TI berfungsi sebagai mekanisme pengarah yang menjembatani kepentingan manajemen, tim teknis, dan pemangku kepentingan lainnya.

Dalam konteks ini, governance TI tidak hanya berorientasi pada pengendalian, tetapi juga pada penciptaan nilai. Kerangka governance membantu memastikan investasi teknologi memberikan manfaat terukur dan mengurangi keputusan ad hoc melalui struktur pengambilan keputusan yang konsisten.

Elemen utama governance TI meliputi:

- Struktur pengambilan keputusan, yang menetapkan peran, tanggung jawab, dan wewenang.

- Kebijakan dan standar, untuk mengatur praktik pengembangan, keamanan, dan pengelolaan data.
- Manajemen risiko dan kepatuhan, guna memastikan kesesuaian dengan regulasi dan kebijakan internal.
- Pengukuran kinerja TI, untuk menilai kontribusi teknologi terhadap tujuan organisasi.

Implementasi governance TI harus fleksibel dan adaptif, terutama dalam lingkungan Agile dan DevOps, agar tidak menghambat inovasi. Dengan governance TI yang kuat dan terintegrasi, organisasi mampu mengelola kompleksitas proyek skala besar secara berkelanjutan serta menjaga keselarasan antara keputusan teknis dan strategi bisnis jangka panjang.

F. Kematangan Proses Rekayasa Perangkat Lunak

Kematangan proses rekayasa perangkat lunak menggambarkan sejauh mana sebuah organisasi mampu mengelola, mengendalikan, dan meningkatkan proses pengembangan perangkat lunaknya secara konsisten. Pada proyek perangkat lunak skala besar, kematangan proses menjadi faktor penentu keberhasilan jangka panjang karena kompleksitas sistem menuntut proses yang stabil, terukur, dan dapat direplikasi. Organisasi dengan tingkat kematangan rendah cenderung bergantung pada individu, sementara organisasi yang matang mengandalkan proses yang terdokumentasi dan terstandarisasi.

Kematangan proses tidak hanya berkaitan dengan dokumentasi atau kepatuhan prosedur, tetapi juga dengan kemampuan organisasi dalam belajar dan beradaptasi. Proses yang matang memungkinkan organisasi mengidentifikasi kelemahan, melakukan perbaikan berkelanjutan, serta merespons perubahan teknologi dan kebutuhan

bisnis dengan lebih terstruktur. Dalam proyek skala besar, hal ini sangat penting untuk menjaga konsistensi kualitas di tengah pertumbuhan tim dan sistem.

Secara umum, kematangan proses rekayasa perangkat lunak dapat dipahami melalui beberapa karakteristik utama:

- Standarisasi proses, di mana praktik pengembangan diterapkan secara konsisten di seluruh tim.
- Pengukuran dan evaluasi, yang memungkinkan kinerja proses dinilai berdasarkan data.
- Pengendalian kualitas, melalui pengujian, audit proses, dan umpan balik berkelanjutan.
- Perbaikan berkelanjutan, dengan menggunakan hasil evaluasi untuk meningkatkan proses secara sistematis.

Pada tingkat kematangan yang lebih tinggi, organisasi tidak hanya mampu menjalankan proyek dengan lebih prediktif, tetapi juga meningkatkan efisiensi dan keandalan sistem yang dihasilkan. Risiko proyek dapat dikelola lebih baik karena proses yang matang menyediakan mekanisme deteksi dini terhadap potensi masalah. Selain itu, estimasi biaya dan waktu menjadi lebih akurat karena didukung oleh data historis dan proses yang terukur.

BAB XII
MANAJEMEN PROYEK PERANGKAT LUNAK SKALA BESAR

BAB ini membahas penerapan nyata arsitektur perangkat lunak modern dalam berbagai sektor industri untuk menjembatani teori dan praktik rekayasa perangkat lunak skala besar. Melalui studi kasus pada sistem e-commerce, keuangan digital, kesehatan, dan smart city, BAB ini menunjukkan bagaimana keputusan arsitektur diambil untuk menjawab kebutuhan skalabilitas, keandalan, keamanan, dan kinerja dalam kondisi operasional yang kompleks. Analisis difokuskan pada tantangan implementasi, solusi yang diterapkan, serta evaluasi arsitektur sistem nyata agar pembaca memperoleh pemahaman praktis tentang bagaimana prinsip rekayasa perangkat lunak tingkat lanjut diwujudkan dalam sistem industri yang berfungsi dan berkelanjutan.

A. Arsitektur Sistem E-Commerce Skala Besar

Platform e-commerce skala besar seperti **Shopee** dan **Amazon** merupakan contoh nyata penerapan arsitektur perangkat lunak modern untuk melayani jutaan hingga ratusan juta pengguna secara simultan. Sistem e-commerce tidak hanya memproses transaksi jual beli, tetapi juga menangani pencarian produk, rekomendasi, pembayaran digital, logistik, promosi real-time, serta analitik data berskala masif. Kompleksitas kebutuhan ini mendorong adopsi arsitektur terdistribusi dan berorientasi layanan agar sistem tetap responsif dan mudah dikembangkan.

Secara evolusioner, arsitektur e-commerce skala besar bergerak dari monolitik menuju *microservices architecture*. Pada Shopee dan

Amazon, fungsi inti seperti katalog produk, keranjang belanja, pembayaran, pengiriman, dan rekomendasi dipisahkan menjadi layanan independen. Pemisahan ini memungkinkan setiap layanan dikembangkan, diuji, dan diskalakan secara terpisah sesuai karakteristik bebannya. Misalnya, layanan pencarian dan promosi cenderung mengalami lonjakan trafik ekstrem saat kampanye besar, sementara layanan lain relatif stabil.

Dari sisi skalabilitas dan kinerja, platform e-commerce memanfaatkan *load balancing*, *auto-scaling*, serta *caching* agresif. Konten statis didistribusikan melalui *Content Delivery Network* untuk menekan latensi, sedangkan backend menggunakan replikasi dan *sharding* basis data agar mampu menangani volume transaksi tinggi. Aspek keandalan dijaga melalui *fault tolerance*, *circuit breaker*, dan *retry mechanism*, sehingga kegagalan satu layanan tidak melumpuhkan sistem secara keseluruhan. Arsitektur ini juga mempercepat inovasi bisnis, memungkinkan peluncuran fitur dan eksperimen secara bertahap dengan risiko teknis yang terkontrol.

B. Sistem Keuangan Digital dan Fintech

Sistem keuangan digital dan fintech merupakan salah satu domain paling menantang dalam penerapan arsitektur perangkat lunak modern karena beroperasi pada area yang sangat sensitif terhadap keamanan, keandalan, dan kepatuhan regulasi. Platform seperti PayPal, Alipay, serta berbagai layanan dompet digital, *mobile banking*, dan *buy now pay later* harus memproses transaksi finansial secara *real-time* dengan toleransi kesalahan yang sangat rendah. Kegagalan sistem pada konteks ini tidak hanya berdampak teknis, tetapi juga berimplikasi langsung pada kepercayaan pengguna, risiko hukum, dan reputasi institusi keuangan.

Secara arsitektural, sistem fintech modern umumnya menerapkan arsitektur terdistribusi berbasis layanan dengan pemisahan ketat antara layanan inti keuangan dan layanan pendukung. Layanan autentikasi, manajemen akun, pemrosesan transaksi, deteksi fraud, dan pelaporan keuangan diisolasi untuk meningkatkan ketahanan sistem dan kemudahan audit. Keamanan menjadi fondasi utama melalui penerapan *defense in depth*, enkripsi menyeluruh, autentikasi berlapis, serta prinsip *zero trust*. Dari sisi kinerja dan skalabilitas, sistem fintech memanfaatkan pemrosesan asinkron dan *event-driven architecture* untuk menangani lonjakan transaksi, sambil tetap memprioritaskan konsistensi data yang kuat demi menjaga akurasi dan integritas transaksi keuangan.

C. Sistem Kesehatan Digital

Sistem kesehatan digital merupakan salah satu domain paling kompleks dalam penerapan arsitektur perangkat lunak modern karena berkaitan langsung dengan keselamatan pasien, kerahasiaan data medis, dan kepatuhan regulasi yang ketat. Platform seperti *health information systems*, *electronic health records* (EHR), telemedicine, dan sistem manajemen rumah sakit harus mampu mengintegrasikan data dari dokter, laboratorium, apotek, asuransi, hingga perangkat medis. Contoh implementasi berskala besar dapat dilihat pada platform EHR seperti Epic Systems dan Cerner, yang digunakan oleh ribuan fasilitas kesehatan dan melayani jutaan rekam medis secara terpusat dan berkelanjutan.

Dari sisi arsitektur, sistem kesehatan digital umumnya dirancang secara modular dan terdistribusi untuk memisahkan fungsi seperti rekam medis, penjadwalan, *billing*, manajemen farmasi, dan analitik klinis. Pendekatan ini meningkatkan keandalan sekaligus

mempermudah integrasi dengan sistem eksternal. Keamanan dan privasi menjadi fondasi utama melalui enkripsi menyeluruh, kontrol akses berbasis peran, audit log yang ketat, serta autentikasi kuat sesuai regulasi. Selain itu, sistem dirancang dengan *high availability*, replikasi data, dan mekanisme pemulihan bencana agar layanan tetap berjalan hampir tanpa henti, mengingat gangguan sistem dapat berdampak langsung pada kualitas dan keselamatan pelayanan kesehatan.

D. Smart City dan Internet of Things

Smart city merupakan contoh penerapan arsitektur perangkat lunak skala besar yang mengintegrasikan sistem digital, infrastruktur fisik, dan data real-time untuk meningkatkan kualitas hidup masyarakat. Konsep ini memanfaatkan Internet of Things (IoT) untuk menghubungkan berbagai perangkat seperti sensor lalu lintas, kamera pengawas, meter pintar, dan layanan publik ke dalam satu ekosistem terpadu. Implementasi smart city dapat ditemukan di kota-kota seperti Singapore dan Barcelona, yang menggunakan teknologi digital untuk meningkatkan efisiensi layanan, transparansi, dan pengambilan keputusan berbasis data.

Dari sisi arsitektur, sistem smart city dibangun secara terdistribusi dan berbasis event, dengan pemrosesan awal dilakukan di *edge devices* sebelum data dikirim ke platform pusat untuk analitik lanjutan. Pendekatan ini menekan latensi dan memungkinkan respons cepat, misalnya pada pengaturan lalu lintas adaptif atau pemantauan lingkungan. Tantangan utama meliputi skalabilitas, interoperabilitas antar sistem dari berbagai vendor, serta keamanan dan privasi data warga. Untuk itu, arsitektur smart city mengandalkan API terbuka,

message broker, segmentasi jaringan, dan prinsip *privacy by design* agar sistem tetap aman, fleksibel, dan berkelanjutan.

E. Evaluasi Arsitektur Sistem Nyata

Evaluasi arsitektur sistem nyata bertujuan menilai apakah desain arsitektur yang diterapkan benar-benar mampu mendukung kebutuhan teknis, bisnis, dan operasional dalam kondisi riil. Berbeda dari evaluasi konseptual, evaluasi ini mempertimbangkan beban pengguna aktual, perubahan kebutuhan, keterbatasan infrastruktur, serta dinamika organisasi. Fokus utamanya adalah memastikan arsitektur tetap relevan dan berkelanjutan dalam jangka panjang.

Dalam praktik, evaluasi arsitektur difokuskan pada *quality attributes* utama seperti kinerja, skalabilitas, keandalan, keamanan, dan keterpeliharaan. Setiap domain memiliki penekanan berbeda, sehingga evaluasi bersifat kontekstual. Aktivitas evaluasi umumnya meliputi:

- Analisis kinerja operasional, untuk menilai respons sistem pada beban nyata.
- Penilaian risiko dan kegagalan, berdasarkan insiden yang pernah terjadi.
- Evaluasi evolvabilitas, terkait kemampuan sistem beradaptasi terhadap perubahan.
- Umpan balik pengguna dan pemangku kepentingan, sebagai indikator non-teknis.

Hasil evaluasi biasanya mengungkap trade-off antar atribut kualitas. Tujuannya bukan mencari arsitektur sempurna, melainkan

memahami kompromi yang ada dan menilai apakah masih dapat diterima sesuai konteks bisnis dan operasional sistem.

DAFTAR PUSTAKA

API7.ai. (2025). *API gateway and service discovery*. https://api7.ai/blog/api-gateway-and-service-discovery

Bass, L., Clements, P., & Kazman, R. (2022). *Software architecture in practice* (4th ed.). Addison-Wesley.

Baxter, G., & Sommerville, I. (2011). Socio-technical systems: From design methods to systems engineering. *Interacting with Computers, 23*(1), 4–17. https://doi.org/10.1016/j.intcom.2010.07.003

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media.

Bondi, A. B. (2014). *Foundations of software and system performance engineering: Process, performance modeling, requirements, testing, scalability, and practice*. Addison-Wesley.

Brewer, E. A. (2012). CAP twelve years later: How the "rules" have changed. *Computer*, 45(2), 23–29.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*, 59(5), 50–57.

Ferraiolo, D. F., Kuhn, D. R., & Chandramouli, R. (2016). *Role-based access control* (2nd ed.). Artech House.

Fowler, M., & Humble, J. (2010). *Continuous delivery*. Addison-Wesley.

Fowler, M., & Lewis, J. (2014). *Microservices: A definition of this new architectural term*. ThoughtWorks.

Gao, J., Tsao, H.-S. J., & Wu, Y. (2015). *Testing and quality assurance for component-based software*. Artech House.

Google. (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media.

Grafana Labs. (n.d.). *Grafana documentation*. https://grafana.com/docs/

Gregg, B. (2020). *Systems performance: Enterprise and the cloud* (2nd ed.). Pearson.

Hardt, D. (2012). The OAuth 2.0 authorization framework. *IETF RFC 6749*. Internet Engineering Task Force.

Hilty, L. M., & Aebischer, B. (2015). ICT for sustainability: An emerging research field. *ICT Innovations for Sustainability*, 3–36. https://doi.org/10.1007/978-3-319-09228-7_1

Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.

Khan, S. U., Ahmad, I., & Ranjan, R. (2015). Energy-efficient resource allocation in cloud computing: A survey. *Computing*, 98(4), 423–457.

Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The DevOps handbook*. IT Revolution Press.

Kleppmann, M. (2017). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.

Koomey, J. G. (2011). Growth in data center electricity use 2005 to 2010. *Analytics Press*.

Lago, P., Koçak, S. A., Crnkovic, I., & Penzenstadler, B. (2021). Framing sustainability as a property of software quality. *Communications of the ACM, 64*(10), 70–78. https://doi.org/10.1145/3460779

Li, D., Chen, Y., Tran, T., & Katz, R. (2013). Energy proportional computing: A survey. *IEEE Computer*, 46(9), 12–21.

Liu, Z., Lin, M., Wierman, A., Low, S. H., & Andrew, L. L. H. (2012). Greening geographical load balancing. *IEEE/ACM Transactions on Networking*, 20(2), 657–671.

Mell, P., & Grance, T. (2011). *The NIST definition of cloud computing*. National Institute of Standards and Technology.

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239).

Mishra, A., Hellerstein, J. L., Cirne, W., & Das, C. R. (2010). Towards characterizing cloud backend workloads: Insights from Google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4), 34–41.

Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2nd ed.). O'Reilly Media.

NIST. (2020). *Zero trust architecture* (Special Publication 800-207). National Institute of Standards and Technology.

Otto, B. (2011). Organizing data governance: Findings from the telecommunications industry and consequences for large service providers. *Communications of the Association for Information Systems*, 29(1), 45–66.

OWASP Foundation. (2023). *OWASP API security top 10*. OWASP Foundation.

PagerDuty. (2021). *Incident response guide*. https://www.pagerduty.com/resources/learn/incident-response/

Pahl, C. (2015). Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3), 24–31.

Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.

Prometheus Authors. (n.d.). *Prometheus documentation* https://prometheus.io/docs/

Red Hat. (n.d.). *What is observability?* https://www.redhat.com/en/topics/devops/what-is-observability

Richards, M., & Ford, N. (2020). *Fundamentals of software architecture: An engineering approach*. O'Reilly Media.

Richardson, C. (2018). *Microservices patterns: With examples in Java*. Manning Publications.

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). A comprehensive study of convergent and commutative replicated data types. *INRIA Research Report*.

Shostack, A. (2014). *Threat modeling: Designing for security*. Wiley.

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education.

Sommerville, I., Sawyer, P., & Vyas, N. (2012). Requirements engineering for systems of systems. *Proceedings of the IEEE, 101*(1), 19–33. https://doi.org/10.1109/JPROC.2012.2198841

Stonebraker, M., & Cetintemel, U. (2005). "One size fits all": An idea whose time has come and gone. *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2–11.

Storey, M. A., Zagalsky, A., Filho, F., Singer, L., & German, D. M. (2020). How social and communication channels shape and challenge

software development. *IEEE Software*, *37*(1), 92–98. https://doi.org/10.1109/MS.2019.2941615

Stuttard, D., & Pinto, M. (2018). *The web application hacker's handbook* (2nd ed.). Wiley.

Turnbull, J. (2018). *The site reliability workbook*. O'Reilly Media.

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1), 40–44.

Weyns, D., Andersson, J., Malek, S., & Schmerl, B. (2012). On patterns for decentralized control in self-adaptive systems. *Software Engineering for Self-Adaptive Systems II*, 76–107. https://doi.org/10.1007/978-3-642-35813-5_4

Wieder, P., Butler, J. M., Theilmann, W., & Yahyapour, R. (2013). Service level objectives in cloud computing. *Proceedings of the IEEE*, 101(7), 1510–1525.

Zhang, Y., Chen, D., & Li, J. (2019). Security and privacy in cloud computing: A survey. *Journal of Network and Computer Applications*, 130, 94–110.

BIOGRAFI PENULIS

Ir. Soleman, S.Kom., M.Kom., MCE.

merupakan akademisi dan praktisi di bidang Teknologi Informasi dan Rekayasa Komputer. Lulusan Sarjana Komputer konsentrasi Sistem Informasi, Magister Ilmu Komputer konsentrasi Teknologi Sistem Informasi, serta Profesi Keinsinyuran bidang *Computer Science & Engineering,* dan memiliki sertifikasi internasional *Microsoft Certified Educator* (MCE). Saat ini aktif sebagai Dosen Tetap Universitas Borobudur, Pengurus Asosiasi Riset Teknik Elektro dan Informatika Indonesia (ARTEII) Divisi Pengembang Organisasi periode 2024–2029, serta Anggota APTIKOM dan Anggota Asosiasi Pengelola Jurnal Indonesia (APJI), serta berperan sebagai reviewer pada jurnal nasional terakreditasi JITK (Sinta 2), PILAR (Sinta 3), dan INFOTECH (Sinta 4).

Dwi Retnoningsih, ST, MT.

Lulus S2 di Program Studi Teknik Elektro, Fakultas Teknik, Universitas Gadjah Mada Yogyakarta. Lulus S1 di Program Studi Manajemen Informatika dan Teknik Komputer, Fakultas Teknologi Industri, Institut Sains dan Teknologi "Akprind" Yogyakarta. Saat ini sebagai dosen tetap di Program Studi Informatika, Fakultas Sains, Teknologi, dan Kesehatan (FSTK), Universitas Sahid Surakarta. Bidang riset *database, Software engineering, web programming, mobile programming, Software Quality Assurance* (SQA). Menjadi anggota

organisasi profesi *Indonesian Computer Electronics and Instrumentation Support Society*-IndoCEISS (2022-2023), sebagai anggota organisasi profesi Asosiasi Riset Teknik Elektro dan Informatika Indonesia-ARTEII (2024-2029), sebagai anggota Asosiasi Pengelola Jurnal Indonesia-APJI (2024-2029).

Ir. Ade Davy Wiranata, S.Kom., M.Kom.

adalah dosen tetap di Universitas Muhammadiyah Prof. Dr. HAMKA (UHAMKA) meraih Sarjana Teknik Informatika dari Universitas Islam Kalimantan (2016) dan Magister Ilmu Komputer dari Universitas Budi Luhur (2019). Saat ini menjabat sebagai Koordinator Merdeka Belajar Kampus Merdeka (MBKM) di Fakultas Teknologi Industri dan Informatika (FTII), berpengalaman sebagai Pengajar Program Prakerja di platform MojadiPro, serta aktif sebagai anggota APTIKOM, APSI- PTMA, dan Asosiasi Dosen Indonesia (ADI).

Hardika Khusnuliawati, S.Kom., M.Kom.

Lulus S1 dan S2 Program Studi Teknik Informatika dari Institut Teknologi Sepuluh Nopember (ITS). Merupakan dosen dan peneliti di bidang Informatika dengan fokus keilmuan pada kecerdasan buatan, *pattern recognition, dan business intelligence*. Aktif melakukan penelitian yang menitikberatkan pada penerapan algoritma *machine learning*, pengenalan pola, serta analitik data untuk mendukung sistem pendukung keputusan. Sejumlah hasil

penelitiannya telah dipublikasikan pada jurnal ilmiah bereputasi nasional dan internasional, yang membahas klasifikasi data, analisis prediktif, dan pemodelan sistem cerdas pada berbagai domain aplikasi. Saat ini, mengabdi sebagai dosen di Universitas Sahid Surakarta dan berkontribusi dalam pengembangan riset serta pendidikan berbasis data dan kecerdasan buatan.

Rekayasa Perangkat Lunak Tingkat Lanjut Arsitektur Modern dan Skalabilitas Sistem

Buku ini menyajikan pembahasan komprehensif mengenai transformasi rekayasa perangkat lunak dari sistem konvensional menuju arsitektur modern yang kompleks, terdistribusi, dan berorientasi layanan. Dimulai dari landasan konseptual rekayasa perangkat lunak tingkat lanjut, buku ini mengulas secara mendalam prinsip-prinsip arsitektur perangkat lunak modern, hubungan antara arsitektur, kinerja, dan skalabilitas, serta peran perangkat lunak sebagai sistem sosio-teknis dalam konteks global. Pembaca diajak memahami perubahan paradigma dari monolithic menuju service-oriented architecture dan microservices, diikuti dengan pembahasan arsitektur cloud-native, containerization, orkestrasi, serta praktik continuous integration dan continuous deployment sebagai tulang punggung pengembangan sistem modern.

Pada bagian lanjutan, buku ini mengupas strategi skalabilitas, manajemen data terdistribusi, keamanan sistem, hingga observability dan reliability engineering sebagai fondasi ketahanan sistem skala besar. Aspek rekayasa kinerja, optimasi energi, serta penerapan green computing dibahas untuk menjawab tantangan efisiensi dan keberlanjutan. Buku ini juga mengeksplorasi integrasi kecerdasan buatan dalam arsitektur perangkat lunak melalui MLOps, otomasi pengujian, dan sistem self-healing. Ditutup dengan pembahasan manajemen proyek skala besar, studi kasus industri, serta proyeksi masa depan arsitektur perangkat lunak termasuk edge computing, quantum computing, dan sustainability, buku ini dirancang sebagai referensi strategis bagi mahasiswa, dosen, peneliti, praktisi TI, dan pengambil kebijakan dalam merancang sistem perangkat lunak yang andal, adaptif, dan berdaya saing tinggi.

9 786347 483966
IKAPI
No. 44/ SBA / 2023
Google Play Books
Takaza