Analisis Soal UTS AAT (IF4031)

Analisis soal tiga tahun, kerangka menjawab studi kasus,
jawaban semua soal lama, dan soal prediksi

Dokumen ini adalah pendamping catatan *UTS_AAT* dan *Glosarium_AAT*. Isinya empat bagian. Pertama, analisis soal UTS tiga tahun (2022/2023, 2023/2024, 2025/2026), dengan bobot lebih besar untuk soal yang lebih baru. Kedua, dua kerangka menjawab, yaitu untuk studi kasus desain arsitektur dan untuk soal troubleshooting. Ketiga, jawaban semua soal lama. Keempat, soal prediksi beserta jawabannya.

Sumber soal. Soal 2023/2024 dan 2025/2026 ditulis ulang dari ingatan oleh mahasiswa (nomor 1 tahun 2023/2024 terlupa), jadi kalimatnya bisa sedikit berbeda dari aslinya. Soal 2022/2023 berupa foto kertas soal asli.

Cakupan. Cakupan UTS sama dengan catatan *UTS_AAT*, yaitu **note 01–08**. Dua soal 2025/2026 (mitigasi database overload dan studi kasus arsitektur) paling lengkap dibahas di note 10 dan 17, yang di luar cakupan itu. Keduanya tetap dijawab dengan konsep note 01–08, dan materi dari note 10/17 ditandai *(note 10/17)* sebagai pelengkap. Penjelasan yang tidak ada di catatan ditandai *(tambahan)*.

Cara membaca jawaban. Tiap jawaban berisi tiga lapis:

- **Intinya**: satu-dua kalimat untuk diingat;
- **Penjelasan**: supaya paham alasannya;
- **Versi ujian**: yang ditulis di lembar jawaban.

Rujukan "UTS 2.x" menunjuk ke bagian catatan *UTS_AAT*.

Daftar Isi

1 Analisis Soal **3**

1.1 Daftar dan Pemetaan Soal 3

1.2 Pola dan Arah Perubahan 3

1.3 Prioritas Belajar 4

2 Dua Kerangka Menjawab **4**

2.1 Kerangka Studi Kasus Desain Arsitektur 4

2.2 Kerangka Soal Troubleshooting/Mitigasi 5

3 Soal Lama dan Jawaban: UTS 2025/2026 **6**

3.1 Nomor 1. Mitigasi database overload pada website beasiswa 6

3.2 Nomor 2. Tujuan memisahkan build, release, run (12-factor app) 7

3.3 Nomor 3. Trade-off layered architecture vs microservice 7

3.4 Nomor 4. Studi kasus: scanner kartu kampus dengan RBAC 8

4 Soal Lama dan Jawaban: UTS 2023/2024 **10**

4.1 Nomor 2. Pertimbangan memilih teknologi message queue 10
4.2 Nomor 3. CQRS dan Saga 11
4.3 Nomor 4. REST vs RPC 12
4.4 Nomor 5. Statefulness dan scalability 13

5 Soal Lama dan Jawaban: UTS 2022/2023 **14**
5.1 Nomor 1. Middleware 14
5.2 Nomor 2. Lima model I/O 15
5.3 Nomor 3. Perbandingan REST API, gRPC, GraphQL 16
5.4 Nomor 4. API Gateway vs Service Mesh 16

6 Soal Prediksi dan Jawaban **17**
6.1 P1. Studi kasus: sistem war tiket konser 18
6.2 P2. Load balancer: NAT vs reverse proxy, lokal vs global 19
6.3 P3. Server untuk puluhan ribu koneksi (C10K) 19
6.4 P4. Asynchronous programming 19
6.5 P5. Desain REST yang benar dan idempotensi 20
6.6 P6. Memilih mekanisme autentikasi 20
6.7 P7. Memilih API style per kebutuhan 21
6.8 P8. Delivery guarantee dan Kafka consumer group 21
6.9 P9. Routing di RabbitMQ 21
6.10 P10. Ketahanan microservice: fan-out dan service lambat 22
6.11 P11. Migrasi monolith ke microservice 22

7 Ringkasan Inti **23**

1 Analisis Soal

1.1 Daftar dan Pemetaan Soal

Tahun	No	Soal (ringkas)	Topik	Note	UTS_AAT
2025/2026	1	Website beasiswa tidak bisa diakses karena database overload; langkah mitigasi	Skalabilitas, beban, caching, queue	01, 06, 07, 08 (+10, 17)	2.1, 2.4, 2.10–2.12
	2	Tujuan memisahkan build, release, run pada 12-factor app	12-factor app	07	2.9
	3	Trade-off layered architecture vs microservice	Monolith/layered vs microservices	01, 07 (+17)	2.2, 2.9
	4	Studi kasus scanner kartu kampus dengan RBAC (reliable, auth sangat cepat, log, CRUD RBAC; 100.000 mahasiswa, 500 gerbang, 100 KB/mahasiswa); gambarkan arsitektur	Desain arsitektur	01–08 (+17)	semua
2023/2024	1	(terlupa)	–	–	–
	2	Minimal 3 pertimbangan memilih teknologi message queue, dengan ilustrasi	Messaging	08	2.11
	3	CQRS dan Saga, dan bedanya	Pattern microservices	07	2.12
	4	Minimal 3 perbedaan REST dan RPC; mana yang lebih populer dan kenapa	API, RPC	04, 05	2.7, 2.8
	5	Pengaruh statefulness terhadap scalability; ilustrasi scaling aplikasi stateful	Stateless vs stateful	01	2.3
2022/2023	1	Middleware: 3 layanan umum + contoh; 3 jenis + produk; konteks pemakaian tiap jenis	Middleware	01	2.2
	2	5 model I/O, cara kerja, kelebihan dan kekurangan	Model I/O server	02	2.5
	3	Perbandingan REST API, gRPC, GraphQL	RPC modern	04, 05	2.7, 2.8
	4	Fungsi dan perbedaan API Gateway dan Service Mesh	API gateway, service mesh	06	2.10

1.2 Pola dan Arah Perubahan

Bobot. Makin baru, makin besar korelasinya dengan UTS tahun ini, jadi pola 2025/2026 diberi bobot tertinggi.

Arah perubahan bentuk soal:

Tahun	Bentuk dominan	Contoh
2022/2023	**Hafalan terstruktur**: "sebutkan N, jelaskan, kelebihan dan kekurangan"	5 model I/O, 3 jenis middleware
2023/2024	**Perbandingan dan alasan**: "jelaskan perbedaan", "mana yang lebih populer, kenapa", "berikan ilustrasi"	REST vs RPC, CQRS vs Saga
2025/2026	**Penerapan pada kasus**: troubleshooting, trade-off, dan **mendesain arsitektur dari requirement dan angka**	Database overload, scanner kartu kampus

Artinya, UTS tahun ini kemungkinan besar berisi:

1. **Satu studi kasus desain arsitektur** (gambar arsitektur + alasan), seperti scanner kartu 2025/2026.
2. **Satu soal troubleshooting/mitigasi** dari skenario gangguan.
3. **Dua-tiga soal konsep** dalam bentuk "tujuan", "trade-off", atau "perbedaan + ilustrasi".

Topik yang muncul berulang:

- **REST/RPC/gRPC/GraphQL**: 2022 nomor 3 dan 2023 nomor 4.
- **Microservices (pattern dan trade-off)**: 2023 nomor 3, 2025 nomor 2 dan 3, plus API gateway/service mesh 2022 nomor 4.
- **Skalabilitas, state, dan beban**: 2023 nomor 5, 2025 nomor 1 dan 4.
- **Infrastruktur komunikasi** (middleware, messaging, model I/O): 2022 nomor 1 dan 2, 2023 nomor 2.

Yang tidak pernah ditanyakan:

- detail kode (signature `select`, fungsi libev, API JMS, contoh kode RabbitMQ);
- tabel varian Thrift (TCompactProtocol, TFramedTransport), detail frame HTTP/2, nama konfigurasi Kafka;
- angka statistik (Netflix 2 miliar request, kegagalan cluster tahun pertama).

Angka dan detail itu berguna sebagai **contoh di jawaban**, bukan sebagai hafalan.

1.3 Prioritas Belajar

Prioritas	Topik	Alasan
1	Desain arsitektur dari kasus (load balancer, API gateway, cache, message queue, stateless, auth token); mitigasi beban/overload; trade-off monolith/layered vs microservices; 12-factor app	Pola 2025/2026 (bobot tertinggi)
2	REST vs RPC vs gRPC vs GraphQL; CQRS, Saga, circuit breaker; pemilihan message queue (RabbitMQ, Kafka, MQTT); stateless vs stateful	Muncul berulang di 2022 dan 2023
3	Model I/O dan thread vs event, async programming; middleware; API gateway vs service mesh; autentikasi (session, JWT, OAuth)	Pernah keluar di 2022 atau belum pernah, tetapi bahan pendukung soal desain

2 Dua Kerangka Menjawab

2.1 Kerangka Studi Kasus Desain Arsitektur

Pakai urutan ini untuk soal "gambarkan arsitekturnya".

Enam langkah desain arsitektur

1. **Ubah requirement menjadi kebutuhan kualitas.** Contoh:
   - "reliable" berarti redundansi dan tetap jalan saat sebagian gagal;
   - "sangat cepat" berarti data di dekat pemakai (cache) dan tanpa langkah jaringan yang tidak perlu;

- "admin lihat log" berarti log dikumpulkan terpusat;
- "CRUD" berarti ada service dan database pengelola.

2. **Hitung beban dari angka di soal** (back-of-the-envelope): jumlah pengguna, puncak request per detik, ukuran data total, dan data mana yang benar-benar dibutuhkan di jalur cepat. Angka ini menentukan perlu cache, perlu queue, atau cukup satu server.
3. **Pisahkan jalur kritis dari jalur pendukung.** Jalur kritis harus cepat dan tahan gagal; jalur pendukung (log, laporan, admin) boleh asynchronous.
4. **Pilih komponen dan beri alasan** (tabel di bawah).
5. **Gambar arsitektur**: client → (API gateway/load balancer) → service → data store, plus jalur asynchronous (queue → worker/log store).
6. **Tutup dengan trade-off dan skenario gagal**: apa yang terjadi bila satu komponen mati, dan apa yang dikorbankan (konsistensi, biaya, kompleksitas).

Kebutuhan	**Komponen (bagian UTS_AAT)**
Satu pintu masuk, auth, rate limit, routing	API gateway (2.10)
Bagi beban, health check, rolling update	Load balancer (2.4)
Bisa di-scale out bebas	Service **stateless** + state di DB/cache (2.3)
Baca cepat, kurangi beban DB	Cache (mis. Redis) dengan TTL (2.12)
Pekerjaan tidak perlu langsung, redam lonjakan	Message queue/pub-sub (2.11)
Banyak domain, tim, dan pola beban berbeda	Microservices (2.9)
Cegah kegagalan merambat	Circuit breaker, timeout, fallback (2.12)
Identitas dan hak akses	Token (JWT/OAuth), RBAC (2.7)

2.2 Kerangka Soal Troubleshooting/Mitigasi

Urutan jawaban yang rapi untuk "sistem X tidak bisa diakses karena Y overload":

Lima langkah mitigasi

1. **Pastikan penyebab** lewat monitoring: metrik (CPU, koneksi, query lambat), log terpusat, distributed tracing.
2. **Mitigasi segera** (memulihkan layanan): batasi beban masuk (rate limiting, antrean), lindungi komponen yang kelebihan beban (circuit breaker, graceful degradation), tambah kapasitas cepat (scale up).
3. **Perbaikan jangka menengah**: kurangi beban ke komponen bermasalah (cache, query/index, connection pooling), alihkan pekerjaan ke asynchronous (queue + worker), scale out yang stateless.
4. **Perbaikan jangka panjang**: ubah arsitektur (pisah read/write, replika, partisi, microservices), capacity planning, load testing, autoscaling.
5. **Sebutkan trade-off** tiap langkah.

3 Soal Lama dan Jawaban: UTS 2025/2026

3.1 Nomor 1. Mitigasi database overload pada website beasiswa

Soal

Website pendaftaran beasiswa tidak dapat diakses. Setelah troubleshooting, database ternyata overload. Jelaskan langkah mitigasinya.

Intinya

Lindungi database dari lonjakan sekarang juga, lalu kurangi beban yang memang harus ditanggungnya, lalu ubah arsitektur agar beban baca dan tulis bisa dibagi.

Penjelasan

Pendaftaran beasiswa punya pola beban khas: lonjakan besar menjelang tenggat, didominasi **baca** (melihat info dan status) dan **tulis** sekali per pendaftar (submit formulir). Beban yang melebihi kapasitas membentuk antrean dan menaikkan latency sampai layanan tidak bisa diakses (UTS 2.1). Karena database satu-satunya, ia menjadi **bottleneck dan single point of failure** (kasus chess.com pendekatan 2, UTS 2.3).

Versi ujian (kerangka 2.2)

1. **Pastikan penyebab:** lihat metrik database (koneksi aktif, CPU, query lambat); lihat log dan tracing untuk menemukan query atau endpoint paling mahal.
2. **Mitigasi segera (memulihkan layanan):**
   - **Rate limiting di API gateway** agar request masuk tidak melebihi kapasitas database; kelebihan request ditahan di antrean ("ruang tunggu") atau mendapat respons "coba lagi nanti" (UTS 2.10).
   - **Circuit breaker dan graceful degradation**: fitur non-penting (mis. statistik pendaftar) dimatikan sementara agar fitur inti tetap jalan (UTS 2.12).
   - **Scale up database** sementara (CPU/RAM lebih besar) bila memungkinkan.
3. **Kurangi beban ke database:**
   - **Cache** (mis. Redis) untuk data yang sering dibaca dan jarang berubah (info beasiswa, daftar program), dengan TTL sesuai toleransi data basi; konten statis lewat **CDN** (UTS 2.4, 2.12).
   - **Optimasi query dan index**, serta **connection pooling** supaya jumlah koneksi ke database terkendali *(tambahan)*.
   - **Proses submit secara asynchronous**: formulir dimasukkan ke **message queue**, lalu worker menulis ke database sesuai kecepatan yang sanggup ditanggung database (*load leveling*). Pendaftar langsung mendapat bukti "pendaftaran diterima" (UTS 2.11).
   - **Server aplikasi stateless** di belakang load balancer agar bisa di-scale out tanpa menambah state (UTS 2.3).
4. **Perbaikan arsitektur jangka panjang:**
   - **Pisahkan baca dan tulis**: read replica untuk query baca, primary untuk tulis *(note 10/17)*, atau CQRS (UTS 2.12).
   - **Partisi/sharding** bila data sangat besar *(note 10/17)*.
   - **Load testing dan capacity planning** sebelum periode pendaftaran *(note 10/17)*, plus **autoscaling** (UTS 2.12).

5. **Trade-off:** cache dan read replica membuat data bisa sedikit basi (*stale*); pemrosesan asynchronous membuat status pendaftaran tidak langsung final; rate limiting berarti sebagian pengguna diminta menunggu.

3.2 Nomor 2. Tujuan memisahkan build, release, run (12-factor app)

Soal

Apa tujuan dibedakannya build, release, dan run pada 12-factor app?

Intinya

Agar yang dijalankan di produksi selalu **artefak yang sama dan tidak diubah**, konfigurasinya terpisah, dan setiap rilis bisa dilacak serta di-rollback.

Penjelasan

Catatan hanya menyebut "pisahkan tegas tahap build dan run". Rincian ketiga tahap berikut dari 12factor.net *(tambahan)*:

- **Build:** kode dari repository diubah menjadi artefak yang bisa dijalankan (kompilasi, ambil dependency, aset).
- **Release:** artefak build digabung dengan **config** environment tertentu (faktor 3: config di environment), lalu diberi ID rilis unik.
- **Run:** rilis dijalankan sebagai proses di environment target.

Seperti memasak di restoran: resep dimasak di dapur (build), diberi label meja dan pesanan (release), lalu disajikan (run). Pelayan tidak boleh mengubah masakan di meja pelanggan.

Versi ujian

1. **Tidak ada perubahan kode saat runtime.** Perbaikan selalu lewat build baru, sehingga yang berjalan selalu bisa ditelusuri ke versi kodenya.
2. **Artefak yang sama untuk semua environment.** Satu build dipakai di staging dan production; bedanya hanya config. Ini mendukung **dev/prod parity** (faktor 10) dan mengurangi bug "jalan di staging, gagal di production".
3. **Rollback mudah.** Tiap release punya ID; kembali ke rilis sebelumnya cukup menjalankan release lama, tanpa build ulang.
4. **Otomasi dan continuous deployment.** Tahap terpisah mudah dijadikan pipeline otomatis, dan tahap run dibuat sesederhana mungkin agar proses cepat dinyalakan ulang (faktor 9, disposability), misalnya saat scale out.

3.3 Nomor 3. Trade-off layered architecture vs microservice

Soal

Jelaskan trade-off antara layered architecture (business layer, persistence layer, dst.) dan microservice architecture.

Intinya

Layered memecah aplikasi menurut **lapisan teknis** tetapi umumnya tetap **satu aplikasi** (dibangun dan di-deploy sebagai satu unit, satu database). Microservice memecah menurut **kapabilitas bisnis** menjadi **banyak aplikasi kecil** yang di-deploy dan di-scale sendiri.

Penjelasan

- **Layered architecture** (presentation, business, persistence) adalah cara merapikan **monolith**: kode dipisah per lapisan **logis**, tetapi semua lapisan umumnya dibangun dan di-deploy sebagai satu unit dan memakai satu database. Bedakan dengan **3-tier/n-tier** (note 01, 17), yaitu pemisahan **fisik**: tiap tier bisa berjalan di mesin terpisah *(tambahan)*.
- **Microservice** memotong secara **vertikal**: tiap service punya logika dan datanya sendiri untuk satu kapabilitas bisnis (UTS 2.9).
- Analogi: layered seperti gedung dengan lantai per fungsi (lobi, kantor, gudang) yang harus direnovasi bersama; microservice seperti deretan ruko yang masing-masing punya lantai sendiri.

Versi ujian

Aspek	Layered (monolith berlapis)	Microservice
Pengembangan, testing, debugging	Lebih mudah: satu codebase, panggilan antar lapisan in-process	Lebih sulit: banyak service, testing end-to-end butuh seluruh ekosistem
Transaksi dan konsistensi	Transaksi ACID mudah (satu database)	Transaksi lintas service sulit (butuh Saga), **eventual consistency**
Deployment	Seluruh aplikasi di-deploy ulang untuk perubahan kecil; siklus panjang	Tiap service di-deploy sendiri; rilis cepat dan rollback per service
Scaling	Undifferentiated: seluruh aplikasi digandakan walau yang sibuk hanya satu fitur	Selektif per service sesuai bebannya
Teknologi	Seragam (satu bahasa, runtime, build)	Polyglot per service
Tim	Banyak tim di satu codebase; koordinasi dan merge menjadi bottleneck	1 service = 1 tim, otonomi tinggi
Kegagalan	Satu bug atau request lambat bisa menjatuhkan seluruh aplikasi	Fault isolation, **asal** didesain benar (circuit breaker, timeout)
Performa dan operasi	Tanpa overhead jaringan antar komponen; satu tim Ops cukup	Overhead jaringan dan serialisasi; butuh DevOps, service discovery, monitoring, tracing

Kesimpulan trade-off: layered unggul untuk aplikasi sederhana atau tim kecil. Microservice unggul saat organisasi besar, fitur beragam, dan kebutuhan scaling berbeda per fitur, dengan harga kompleksitas sistem terdistribusi ("menambah kompleksitas demi kesederhanaan di level lain").

3.4 Nomor 4. Studi kasus: scanner kartu kampus dengan RBAC

Soal

Kampus memasang scanner kartu untuk masuk ruangan/gedung dengan Role Based Access Control.
Requirement: (1) sistem reliable; (2) authentication dan authorization sangat cepat; (3) admin bisa melihat log; (4) admin bisa CRUD RBAC.
Data: 100.000 mahasiswa, 500 gerbang akses, 100 KB data per mahasiswa.
Gambarkan arsitekturnya.

Intinya

Keputusan buka gerbang harus diambil **di dekat gerbang** dari data izin yang kecil dan di-cache, sehingga cepat dan tetap jalan walau jaringan atau server pusat bermasalah. Log dan perubahan RBAC dialirkan secara **asynchronous** lewat message broker.

Langkah 1: requirement menjadi kebutuhan

- *Reliable:* gerbang tetap bisa memutuskan saat server pusat atau jaringan terganggu; komponen pusat redundan.
- *Auth sangat cepat:* jangan bergantung pada perjalanan jaringan ke database untuk setiap tap.
- *Log:* setiap tap tercatat terpusat tanpa memperlambat gerbang.
- *CRUD RBAC:* ada service admin, dan perubahannya sampai ke semua gerbang dengan cepat.

Langkah 2: hitung beban (asumsi ditulis)

- **100 KB per mahasiswa** (soal menulis "100kb"; diasumsikan kilobyte, termasuk foto dan profil; bila kilobit, totalnya sekitar 1,25 GB dan kesimpulannya tetap sama):

$$100.000 \times 100 \text{ KB} = 10.000.000 \text{ KB} \approx 10 \text{ GB}.$$

  Data sebesar ini tidak perlu ada di gerbang.
- **Data yang dibutuhkan untuk otorisasi** hanya pemetaan kartu → role dan izin per gerbang, sekitar ratusan byte per mahasiswa. Untuk 100.000 mahasiswa totalnya hanya sekitar **puluhan MB**, sehingga muat di memori **setiap controller gerbang**.
- **Beban puncak:** misalnya separuh mahasiswa masuk dalam 30 menit pagi,

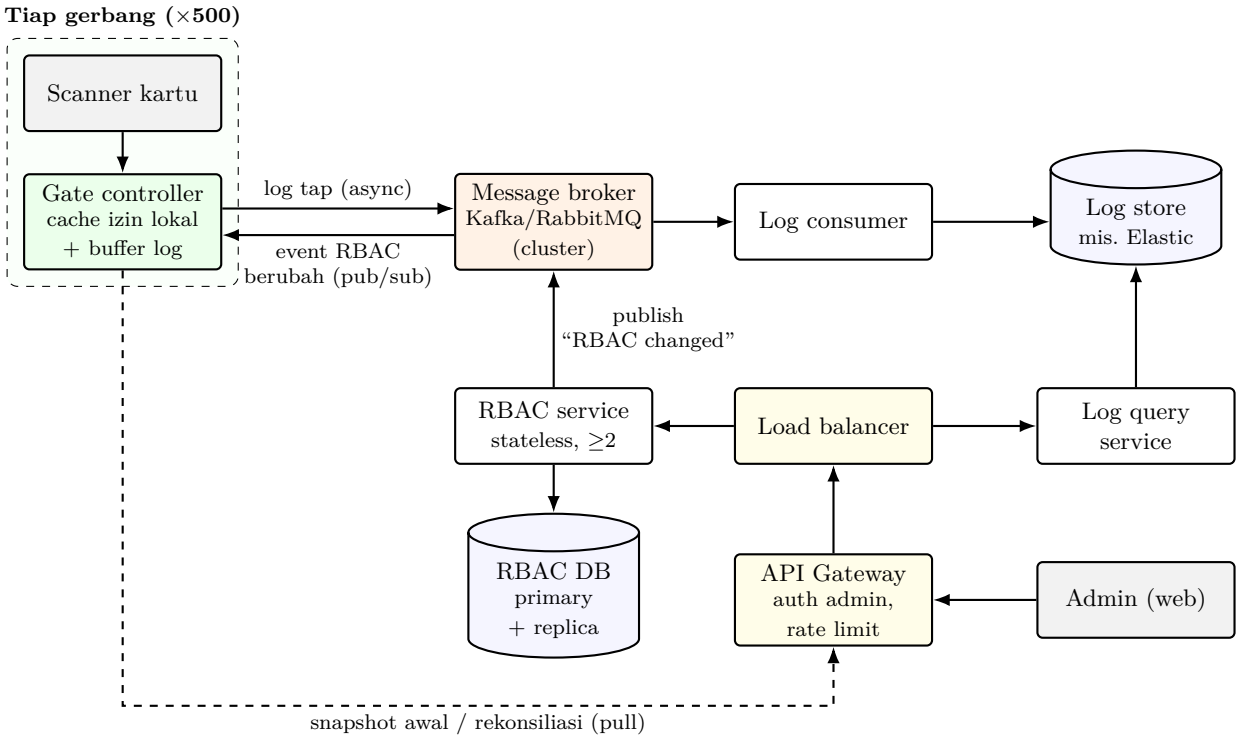
$$\frac{50.000 \text{ tap}}{1.800 \text{ detik}} \approx 28 \text{ tap/detik total} \approx 1 \text{ tap per } 18 \text{ detik per gerbang.}$$

  Throughput kecil, jadi tantangan utamanya **latency dan reliability**, bukan skala.

Langkah 3: jalur kritis vs pendukung

- **Jalur kritis:** tap kartu → putuskan izin → buka gerbang. Harus lokal dan cepat.
- **Jalur pendukung:** kirim log, sinkronisasi perubahan RBAC, dashboard admin. Boleh asynchronous.

Langkah 4–5: arsitektur



Penjelasan komponen:

1. **Gate controller** di tiap gerbang:
   - menyimpan **cache izin lokal** (kartu → role → gerbang yang boleh) di memori, sehingga keputusan buka gerbang diambil **tanpa memanggil server** (milidetik);
   - tetap bekerja saat jaringan putus (mode offline), menyimpan log di **buffer lokal** lalu mengirimnya saat tersambung lagi.
2. **RBAC service** (stateless, minimal dua instance di belakang load balancer): melayani CRUD role dan izin dari admin; menulis ke **RBAC DB** (primary + replica); setiap perubahan **di-publish** sebagai event "RBAC changed".
3. **Message broker** (cluster, direplikasi):
   - **pub/sub** untuk mengirim perubahan RBAC ke 500 gerbang sekaligus (fanout/topic), sehingga cache gerbang segera diperbarui;
   - **queue** untuk log tap, sehingga gerbang tidak menunggu log tersimpan;
   - delivery guarantee log **at least once** dengan consumer idempotent (ID event unik).
4. **Log consumer dan log store:** menyimpan log terpusat yang bisa dicari; admin melihatnya lewat **log query service**.
5. **API gateway:** satu pintu masuk untuk admin (autentikasi admin, otorisasi, rate limiting, logging).
6. **Sinkronisasi awal/rekonsiliasi:** saat gerbang menyala atau tertinggal event, ia mengambil snapshot izin terbaru (pull), lalu menerima perubahan berikutnya lewat pub/sub (push).

Langkah 6: skenario gagal dan trade-off

Kejadian	Akibat
Server pusat atau jaringan mati	Gerbang tetap memutuskan dari cache; log menumpuk di buffer lalu dikirim belakangan
Satu instance RBAC service mati	Load balancer mengalihkan ke instance lain (health check)
Broker satu node mati	Cluster dengan replikasi tetap melayani

Trade-off:

- Pencabutan akses tidak instan bila gerbang sedang offline: ada jendela *eventual consistency* sampai event perubahan diterima. Untuk izin sangat sensitif, bisa ditambahkan **TTL cache pendek** atau pemeriksaan online khusus ruangan kritis *(tambahan)*.
- Data izin tersebar di 500 perangkat, sehingga keamanan perangkat gerbang dan enkripsi cache perlu diperhatikan.

4 Soal Lama dan Jawaban: UTS 2023/2024

Nomor 1 terlupa oleh penulis ulang soal, jadi bagian ini mulai dari nomor 2.

4.1 Nomor 2. Pertimbangan memilih teknologi message queue

Soal

Jelaskan minimal tiga hal yang menjadi pertimbangan dalam memilih teknologi messaging queue, dan berikan ilustrasi pemilihannya.

Intinya

Pilih berdasarkan **pola komunikasi**, **volume dan penyimpanan**, **jaminan pengiriman**, dan **lingkungan perangkat**.

Versi ujian

Pertimbangan	Pertanyaan yang diajukan	Pengaruh pada pilihan
Pola komunikasi dan routing	Butuh point-to-point (satu pesan satu consumer), pub/sub (banyak penerima), atau routing berdasarkan isi?	RabbitMQ kaya routing (exchange direct, fanout, topic); Kafka pub/sub berbasis topic dan consumer group; MQTT pub/sub murni berbasis topic
Volume, throughput, dan retensi	Berapa pesan per detik? Perlu disimpan lama dan bisa di-replay?	Kafka untuk volume sangat tinggi dan durable (log append-only, replay dari offset); RabbitMQ umumnya menghapus pesan setelah di-ack
Delivery guarantee dan reliabilitas	At most once, at least once, atau exactly once? Pesan boleh hilang saat broker mati?	Persistent queue, publisher confirm, manual ack (RabbitMQ); replikasi per partisi (Kafka); QoS 0/1/2 (MQTT)
Urutan pesan	Urutan harus terjaga?	Kafka menjamin urutan hanya **dalam satu partisi**
Lingkungan dan protokol	Client server biasa, atau perangkat IoT berdaya rendah? Perlu standar terbuka?	MQTT ringan untuk IoT; AMQP (RabbitMQ) standar terbuka lintas vendor
Skala dan operasi	Perlu cluster dan high availability? Seberapa rumit dioperasikan?	Kafka (replikasi partisi) vs RabbitMQ (mirroring queue)

Ilustrasi:

- **Aplikasi pemesanan tiket** yang meneruskan permintaan dari frontend ke backend, butuh routing dan request-reply: **RabbitMQ**.
- **Pengumpulan log/clickstream jutaan event per detik** untuk analitik dan bisa diproses ulang: **Kafka**.
- **Sensor suhu di ribuan perangkat IoT** dengan bandwidth kecil: **MQTT**.

4.2 Nomor 3. CQRS dan Saga

Soal

Apa yang dimaksud CQRS dan Saga? Jelaskan perbedaannya.

Intinya

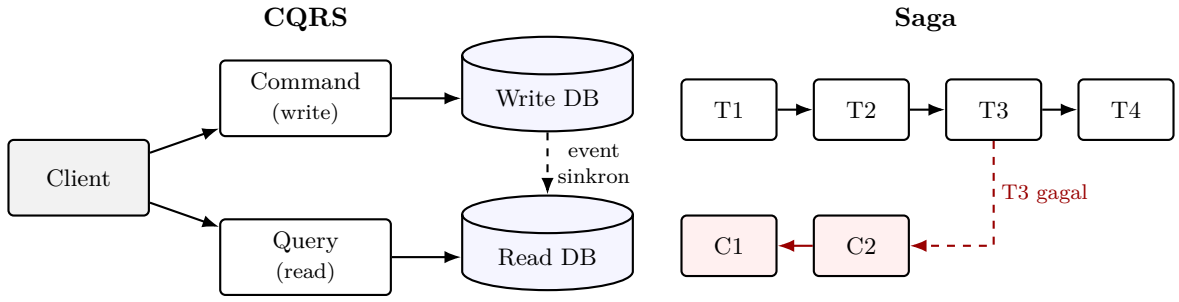
CQRS memisahkan **jalur baca dan tulis**. Saga mengelola **transaksi lintas service**. Keduanya muncul karena microservice memakai database per service, tetapi menyelesaikan masalah yang berbeda.

Penjelasan

Database per service membuat dua hal sulit. Pertama, query yang butuh data dari banyak service, dan beban baca yang jauh lebih besar dari tulis. Kedua, transaksi yang menyentuh beberapa service sekaligus, karena tidak ada transaksi terdistribusi yang murah. CQRS menjawab masalah pertama, Saga menjawab masalah kedua.

Versi ujian

- **CQRS (Command Query Responsibility Segregation):** command (write) masuk ke **write DB**; perubahan disebarkan lewat event ke **read DB**; query membaca dari read DB. Sisi baca (sering didenormalisasi agar cepat) dan sisi tulis (dinormalisasi agar konsisten) bisa dioptimasi dan di-scale sendiri.
- **Saga:** transaksi besar dipecah menjadi **transaksi lokal per service** (T1 → T2 → T3 → T4). Bila satu langkah gagal, **compensating transaction** dijalankan mundur (mis. C2 lalu C1) untuk membatalkan efek langkah sebelumnya. Dua cara: **choreography** (tiap service bereaksi pada event, tanpa koordinator) dan **orchestration** (satu coordinator mengatur urutan).



Aspek	CQRS	Saga
Masalah	Beban dan model baca vs tulis berbeda	Konsistensi transaksi lintas service
Mekanisme	Dua model/database (read dan write) disinkronkan lewat event	Rangkaian transaksi lokal + kompensasi
Hasil	Query cepat, scaling baca/tulis independen	Konsistensi akhir tanpa 2PC
Konsekuensi	Read DB bisa sedikit tertinggal (eventual consistency)	Harus merancang kompensasi; ada keadaan antara
Contoh	Halaman katalog membaca read DB terdenormalisasi	Order → Payment → Stok → Shipment; stok habis membuat pembayaran dikembalikan dan order dibatalkan

4.3 Nomor 4. REST vs RPC

Soal

Jelaskan minimal tiga perbedaan antara REST dan RPC. Mana yang lebih populer? Jelaskan alasannya.

Intinya

REST memodelkan API sebagai **resource** yang dimanipulasi lewat verb HTTP standar. RPC memodelkan API sebagai **pemanggilan fungsi** dengan kontrak eksplisit.

Versi ujian

Aspek	REST	RPC (mis. gRPC, Thrift)
Unit dasar	**Resource** (`/accounts/12345`)	**Aksi/prosedur** (`createMessage`, `getProduct`)
Operasi	Verb HTTP standar (GET, POST, PUT, DELETE) dengan semantik jelas (idempotent kecuali POST)	Nama fungsi bebas sesuai domain
Kontrak	Longgar, didokumentasikan (mis. OpenAPI)	Ketat lewat **IDL** (`.proto`, `.thrift`), kode stub dibangkitkan otomatis
Format dan transport	Umumnya JSON/XML (teks, human-readable) di atas HTTP	Umumnya biner (Protobuf/Thrift Compact), transport HTTP/2 (gRPC) atau TCP (Thrift)
Performa	Overhead lebih besar (header HTTP, parsing teks)	Ringkas dan cepat, cocok untuk pesan sangat banyak
Fitur	Memanfaatkan infrastruktur web (cache, proxy, TLS)	Streaming dua arah (gRPC), evolvability lewat nomor field

Yang lebih populer: REST, terutama untuk API publik, karena:

- hampir semua bahasa punya library HTTP;
- banyak framework siap pakai;
- mudah dibaca dan di-debug;
- bisa memakai cache dan proxy web;
- ukuran pesan bukan masalah utama kebanyakan aplikasi.

RPC (gRPC) makin banyak dipakai untuk komunikasi **internal antar microservice** yang butuh latensi rendah dan pesan sangat banyak. Keduanya bukan saling menggantikan, melainkan dipilih sesuai kebutuhan.

4.4 Nomor 5. Statefulness dan scalability

Soal

Jelaskan pengaruh statefulness sebuah aplikasi terdistribusi terhadap scalability-nya. Berikan contoh ilustrasi scaling pada aplikasi terdistribusi yang stateful.

Intinya

Stateless bisa diperbanyak sebebasnya. Stateful sulit karena state hanya ada di satu tempat, sehingga request harus kembali ke tempat itu dan state hilang bila tempat itu mati.

Penjelasan

Stateless seperti kasir minimarket yang semuanya bisa melayani siapa pun karena data ada di sistem pusat. Stateful seperti dokter yang menyimpan catatan pasien di laci sendiri: pasien harus kembali ke dokter yang sama, dan bila laci hilang, catatannya hilang.

Versi ujian

- **Stateless service** bisa di-**scale out** bebas: instance mana pun bisa melayani request mana pun di belakang load balancer, dan instance bisa ditambah/dikurangi kapan saja.
- **Stateful service** mempersulit scaling:
  1. request harus diarahkan ke instance pemegang state (*affinity*);

2. beban bisa timpang;
3. bila instance mati, state di memorinya hilang;
4. menambah/mengurangi instance memerlukan pemindahan state.

- **Ilustrasi chess.com** (state permainan catur):
  - **Satu server**: semua game di memori satu mesin, tidak scalable.
  - **Scaling stateful naif**: banyak server, tiap game hanya di satu server. Pemain harus selalu diarahkan ke server game-nya, dan bila server mati sekitar $1/n$ game hilang.
  - **Menjadikan stateless**: state game dipindah ke **database bersama**, server mana pun bisa melayani lewat load balancer. Harganya, database menjadi bottleneck dan single point of failure baru.
- **Cara men-scale yang stateful: partisi state per key** (mis. per ID game) dengan routing ke pemiliknya; **replikasi** state agar tidak hilang saat satu node mati; **memindahkan state ke luar service** (database, cache terdistribusi).

5 Soal Lama dan Jawaban: UTS 2022/2023

5.1 Nomor 1. Middleware

Soal

(a) Sebutkan 3 layanan umum middleware dan contoh penggunaannya. (b) Sebutkan 3 jenis middleware dan contoh produknya. (c) Untuk tiap jenis, jelaskan konteks aplikasi yang cocok.

Intinya

Middleware adalah perantara yang menyembunyikan detail komunikasi antar komponen software yang berbeda sistem, dengan fokus integrasi dan interoperabilitas.

Versi ujian

(a) Layanan umum:

- **Transparent location service:** client tidak perlu tahu lokasi server. Contoh: client Java RMI mencari objek remote lewat registry (lookup).
- **Standardized communication:** format dan protokol pertukaran yang disepakati. Contoh: dua aplikasi beda bahasa bertukar pesan lewat AMQP.
- **Network and platform independence:** aplikasi di OS dan bahasa berbeda tetap bisa berkomunikasi. Contoh: client mobile dan server Linux memakai gRPC.
- (Juga: reliability, scalability, availability, mis. broker menyimpan pesan saat penerima mati.)

(b) dan (c) Jenis, produk, dan konteks:

Jenis	Produk	Konteks yang cocok
RPC & serialization	Java RMI, Apache Thrift, Protobuf/gRPC, Avro	Komunikasi sinkron antar service yang butuh pemanggilan seperti fungsi lokal, mis. microservice internal berperforma tinggi
Message-oriented middleware	JMS (ActiveMQ), AMQP (RabbitMQ), Kafka, MQTT	Komunikasi asynchronous dan loosely coupled: pemrosesan latar, event-driven, IoT, pengumpulan log
Application server	JSP server/Tomcat, EJB container	Aplikasi enterprise yang butuh persistensi, transaksi, dan distribusi yang disediakan container
Database access	JDBC/ODBC, ORM (Hibernate, JPA)	Aplikasi yang mengakses basis data tanpa terikat vendor
Enterprise messaging	SOAP, Web Services	Integrasi antar sistem enterprise dengan kontrak formal

5.2 Nomor 2. Lima model I/O

Soal

(a) Sebutkan 5 model I/O yang umum dan cara kerjanya. (b) Jelaskan kelebihan dan kekurangan masing-masing.

Intinya

Semua operasi input punya dua fase: **menunggu data siap** dan **menyalin data ke proses**. Kelima model berbeda pada cara menangani kedua fase itu.

Versi ujian

Model	Cara kerja	Kelebihan	Kekurangan
Blocking	`recvfrom` memblok proses sampai data siap dan selesai disalin	Paling sederhana, tidak membuang CPU	Satu thread hanya melayani satu koneksi; butuh banyak thread, tidak scalable
Non-blocking	`recvfrom` langsung kembali dengan error (`EWOULDBLOCK`) bila belum siap; proses polling berulang	Proses bisa mengerjakan hal lain	Busy-wait membuang CPU; fase salin tetap blok
I/O multiplexing (select/poll/ epoll/kqueue)	Satu panggilan memantau banyak socket; proses blok sampai salah satu siap, lalu `recvfrom`	Satu thread melayani banyak koneksi; tidak busy-wait	select/poll memeriksa linear (mahal untuk banyak socket; epoll/kqueue mengatasinya); fase salin tetap blok

Model	Cara kerja	Kelebihan	Kekurangan
Signal-driven	Daftarkan handler SIGIO; kernel mengirim sinyal saat data siap; proses lalu `recvfrom`	Tidak blok dan tidak busy-wait saat menunggu	Penanganan sinyal rumit, jarang untuk I/O volume tinggi; fase salin tetap blok
Asynchronous (`aio_read`)	Panggilan langsung kembali; kernel mengerjakan kedua fase dan memberi tahu setelah data sudah di buffer proses	Tidak ada blok sama sekali, paling efisien secara teori	Dukungan OS dan API terbatas, pemrograman lebih rumit

Inti perbedaan

Empat model pertama tetap blok saat fase penyalinan. Hanya asynchronous yang tidak blok di kedua fase. Signal-driven memberi tahu saat I/O **bisa dimulai**, asynchronous saat I/O **sudah selesai**.

5.3 Nomor 3. Perbandingan REST API, gRPC, GraphQL

Soal

Jelaskan perbandingan beberapa RPC yang umum digunakan: REST API, gRPC, GraphQL.

Versi ujian

Aspek	REST API	gRPC	GraphQL
Model	Resource + verb HTTP	Pemanggilan prosedur dari kontrak `.proto`	Query/mutation ke skema graf data; client menentukan field
Format dan transport	JSON/XML di atas HTTP (umumnya HTTP/1.1)	Biner (Protobuf) di atas **HTTP/2**	Query teks, response JSON, umumnya di atas HTTP
Endpoint	Banyak endpoint per resource	Satu service dengan banyak method	**Satu endpoint**
Fleksibilitas data	Rendah: rawan over-/under-fetching, data relasional butuh banyak panggilan	Rendah: signature tetap	Tinggi: “ask for what you want, get exactly that”, deep fetch dalam satu request
Performa	Sedang (teks, banyak round-trip)	Tinggi: payload ringkas, multiplexing HTTP/2, streaming dua arah	Mengurangi round-trip, tetapi ada overhead parsing query dan resolver berjenjang
Human-readable	Ya	Tidak	Ya
Cocok untuk	API publik, CRUD sederhana	Komunikasi internal microservice berperforma tinggi dan streaming	Backend aplikasi SPA/mobile dengan data bersarang

5.4 Nomor 4. API Gateway vs Service Mesh

Soal

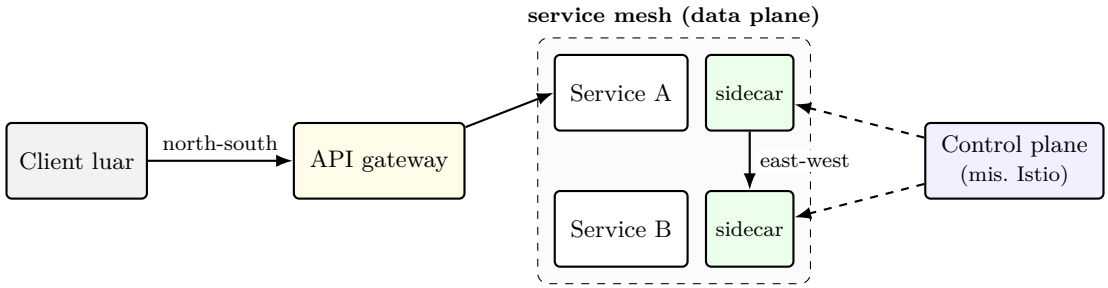
Jelaskan fungsi dan perbedaan antara API Gateway dan Service Mesh.

Intinya

API gateway mengatur lalu lintas **dari luar ke dalam** (north-south). Service mesh mengatur lalu lintas **antar service di dalam** (east-west).

Versi ujian

- **API gateway:** façade/reverse proxy untuk API publik.
  - Fungsi utama: **routing**, **composition** (menggabungkan beberapa API internal), dan **translation** (mis. REST publik ke gRPC internal).
  - Menangani **cross-cutting concern**: autentikasi, otorisasi, rate limiting, timeout/retry, logging, tracing.
  - Mengurangi coupling klien dengan struktur internal, dan mendukung versioning serta monetisasi.
  - Contoh: Kong, AWS API Gateway, Zuul, Traefik.
- **Service mesh:** lapisan infrastruktur yang mengatur komunikasi antar service dari satu **control plane**.
  - Tiap service didampingi **sidecar proxy** (data plane, mis. Envoy) yang menjalankan kebijakan.
  - Fitur: observability, routing (canary), mTLS dan manajemen sertifikat, service discovery, retry, circuit breaking, timeout.
  - Contoh: Istio, Linkerd, Consul.



Aspek	API gateway	Service mesh
Arah trafik	Eksternal → internal	Internal ↔ internal
Letak	Terpusat di tepi sistem	Tersebar sebagai sidecar di samping tiap service, dikendalikan control plane
Fokus	Mengelola API publik (akses, keamanan, komposisi, monetisasi)	Keandalan, keamanan, dan observability komunikasi antar service
Perubahan kode service	Tidak perlu	Tidak perlu (ditangani sidecar)

Keduanya bisa dipakai bersama: gateway untuk pintu masuk, mesh untuk lalu lintas di dalam.

6 Soal Prediksi dan Jawaban

Mengikuti arah 2025/2026, sebagian besar soal prediksi berbentuk **kasus dan trade-off**, memakai topik di note 01–08 yang belum pernah ditanya.

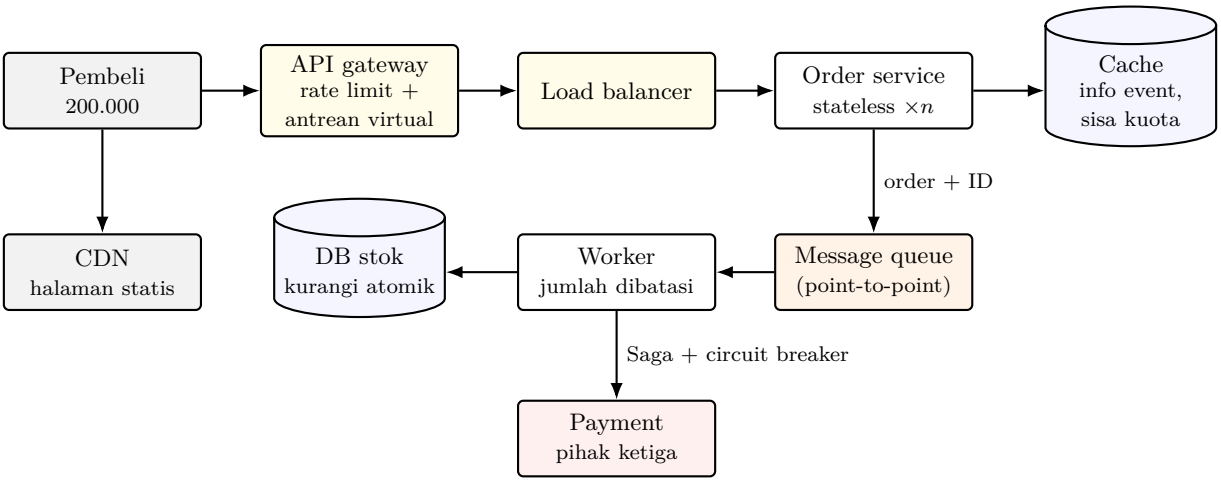
6.1 P1. Studi kasus: sistem war tiket konser

Soal

Penjualan tiket konser dibuka pukul 10.00. Diperkirakan 200.000 orang mengakses bersamaan, sementara kuota hanya 20.000 tiket. Requirement: tidak boleh terjual melebihi kuota, sistem tidak boleh mati, dan pembeli mendapat kepastian. Gambarkan arsitekturnya dan jelaskan.

Jawaban (kerangka 2.1)

- **Kebutuhan:** lonjakan sesaat yang ekstrem; konsistensi stok (tidak oversell); pembeli tidak harus menahan koneksi lama.
- **Hitung beban:** 200.000 orang dalam beberapa menit pertama berarti ribuan request per detik. Mayoritas hanya **membaca** (halaman event, sisa kuota), dan hanya sebagian kecil yang akhirnya **menulis** (pesan tiket).



Arsitektur:

1. **CDN** untuk halaman statis.
2. **API gateway** dengan **rate limiting** dan **antrean virtual**: pengguna diberi nomor urut dan dilepas bertahap sesuai kapasitas.
3. **Load balancer** di depan **service stateless** (event, order) yang di-scale out.
4. **Cache** untuk info event dan perkiraan sisa kuota (TTL pendek).
5. **Pemesanan diproses asynchronous:** order masuk ke **message queue** (point-to-point), lalu **worker** mengurangi stok secara atomik di database. Pembeli mendapat ID order, lalu memantau status (polling atau notifikasi).
6. **Pembayaran** dengan **Saga**: order → reservasi kursi → bayar. Bila pembayaran gagal atau waktu habis, kompensasi melepas kursi.
7. **Circuit breaker** ke layanan pembayaran pihak ketiga.

Trade-off: antrean membuat pengguna menunggu, tetapi sistem tidak runtuh; status order sempat "menunggu" (eventual); jumlah worker yang menulis stok dibatasi agar database tidak overload.

6.2 P2. Load balancer: NAT vs reverse proxy, lokal vs global

Soal

Sebuah layanan web dipakai pengguna di Asia dan Eropa. Jelaskan jenis load balancer yang Anda gunakan dan alasannya.

Jawaban

- **Global (DNS geographic load balancing atau CDN):** mengarahkan pengguna ke data center terdekat agar latency rendah, dan menghindari satu data center menjadi single point of failure. IP Anycast hanya realistis untuk perusahaan yang mengoperasikan jaringan sendiri.
- **Lokal di tiap data center:**

Jenis	Kapan dipakai
Reverse proxy (mis. Nginx)	Layanan HTTP yang butuh SSL termination, caching, dan kompresi; sekitar 100 ribu–1 juta request/detik
NAT (IP layer, translasi IP:port)	Throughput sangat tinggi (1–10 juta request/detik) atau protokol non-HTTP

- **Fitur yang dimanfaatkan:** health check (server gagal dikeluarkan dari rotasi) dan rolling update (update tanpa downtime).
- **Trade-off:** DNS lambat berubah karena TTL cache (menit sampai jam); load balancer lokal sendiri bisa menjadi single point of failure, sehingga perlu redundansi.

6.3 P3. Server untuk puluhan ribu koneksi (C10K)

Soal

Server chat harus menahan 50.000 koneksi WebSocket yang sebagian besar menganggur. Model thread-per-connection atau event-driven? Jelaskan.

Jawaban

- **Pilih event-driven** dengan I/O multiplexing (epoll/kqueue), seperti Nginx atau Node.js/libuv.
- **Thread-per-connection:** 50.000 thread$\times$ $\sim$ 1 MB stack $\approx$ 50 GB memori, ditambah biaya context switch, padahal kebanyakan thread hanya menunggu I/O (aplikasi jaringan bersifat I/O bound).
- **Event-driven:** satu event loop per core memantau semua socket dan hanya bekerja saat ada data. Memorinya kecil dan tidak ada context switch per koneksi.
- **Syarat: handler tidak boleh blocking** (pekerjaan berat dialihkan ke thread pool); pada mode edge-triggered, seluruh data yang tersedia harus dibaca dalam satu kesempatan.
- **Trade-off:** kode lebih sulit (state eksplisit, penanganan error rumit) dibanding alur sekuensial thread.

6.4 P4. Asynchronous programming

Soal

Sebuah endpoint asyncio memanggil `time.sleep(2)` dan membaca file besar secara sinkron. Mengapa seluruh server menjadi lambat, dan bagaimana memperbaikinya?

Jawaban

- **Penyebab:** event loop berjalan di satu thread dan tugas menyerahkan kendali secara sukarela lewat `await`. Panggilan blocking memblokir **seluruh loop**, sehingga semua request lain ikut menunggu.
- **Perbaikan:** ganti dengan versi async (`await asyncio.sleep(2)`, library I/O async); alihkan pekerjaan blocking ke executor/thread pool lewat `loop.run_in_executor(None, fungsi)`.
- **Catatan:** async tidak membuat satu pekerjaan lebih cepat; manfaatnya efisiensi sumber daya saat banyak pekerjaan menunggu I/O.

6.5 P5. Desain REST yang benar dan idempotensi

Soal

Sebuah API memakai `POST /inbox/createMessage` dan `DELETE /user/12/feed/posts/latest`. Apa masalahnya dan bagaimana perbaikannya?

Jawaban

- `POST /inbox/createMessage`: gaya RPC, bukan resource. Diulang (retry) bisa membuat pesan ganda karena POST tidak idempotent. Perbaikannya `PUT /inbox/message/{uuid}` dengan ID dari client: identitas eksplisit sehingga retry aman (idempotent).
- `DELETE /user/12/feed/posts/latest`: DELETE seharusnya idempotent. Bila diulang karena timeout, ia menghapus post "terbaru" berikutnya, sehingga lebih dari satu post terhapus. Perbaikannya merujuk ID: `DELETE /user/12/feed/post/{post-id}`.
- **Alasan penting:** proxy dan client bisa mengulang request saat timeout, jadi semantik method HTTP harus dipatuhi.

6.6 P6. Memilih mekanisme autentikasi

Soal

Sistem terdiri dari 10 microservice dan diakses aplikasi web serta mobile. Bandingkan session ID di server dengan token JWT, lalu pilih.

Jawaban

Aspek	**Session ID (random)**	**JWT (rich token)**
Penyimpanan	Server menyimpan pemetaan session → user	Informasi (user, role, expiry) ada di token yang ditandatangani
Tiap request	Harus query penyimpanan session	Cukup verifikasi tanda tangan, tanpa query
Skalabilitas	Penyimpanan session jadi pusat beban	Service tetap stateless, mudah di-scale
Kekurangan	–	Sulit dicabut sebelum kedaluwarsa; ukuran membesar seiring claim (dikirim tiap request)

- **Pilihan untuk banyak microservice: JWT** lewat **OAuth 2.0/OpenID Connect**. API gateway memverifikasi token sekali di depan, dan service tetap stateless.
- **Mitigasi:** masa berlaku pendek + refresh token, dan claim seperlunya.

6.7 P7. Memilih API style per kebutuhan

Soal

Tentukan gaya API untuk: (a) API publik untuk mitra; (b) komunikasi internal antar 50 service dengan jutaan panggilan per menit; (c) aplikasi mobile yang menampilkan profil, postingan, dan komentar sekaligus; (d) streaming data sensor dua arah.

Jawaban

- **(a) REST:** universal, mudah dipahami mitra, cache web, dokumentasi OpenAPI.
- **(b) gRPC:** biner ringkas, HTTP/2 multiplexing, kontrak `.proto` dengan stub otomatis. Thrift juga cocok bila butuh transport fleksibel atau CPU lebih rendah.
- **(c) GraphQL:** satu request mengambil data bersarang sesuai kebutuhan layar, menghindari banyak round-trip dan over-fetching.
- **(d) gRPC bidirectional streaming:** dukungan streaming dua arah native di atas HTTP/2.

6.8 P8. Delivery guarantee dan Kafka consumer group

Soal

Event pembayaran dikirim lewat Kafka ke dua sistem: notifikasi email dan sistem akuntansi yang punya 3 instance. Rancang consumer-nya dan jelaskan jaminan pengiriman yang dipakai.

Jawaban

- **Dua consumer group berbeda:** "notifikasi" dan "akuntansi". Tiap grup menerima salinan semua event (pub/sub antar grup).
- **Di dalam grup akuntansi**, 3 instance berbagi partisi (load balancing). Jumlah partisi minimal 3 agar ketiganya terpakai.
- **Urutan:** pakai key (mis. ID akun) agar event satu akun selalu masuk partisi yang sama.
- **Jaminan at least once:** offset di-commit setelah selesai diproses. Akibatnya event bisa diproses dua kali saat terjadi retry, jadi consumer harus **idempotent** (simpan ID event yang sudah diproses). Exactly once lebih mahal.

6.9 P9. Routing di RabbitMQ

Soal

Sistem log mengirim pesan dengan routing key `region.level`, mis. `asia.error`. Tim A ingin semua error dari semua region, tim B semua log Asia, dan tim C ingin satu salinan setiap log. Exchange apa yang dipakai?

Jawaban

- **Tim A dan B: topic exchange.** Tim A bind dengan pola `*.error`, tim B dengan `asia.*` (atau `asia.#`). Satu pesan `asia.error` masuk ke queue keduanya.
- **Tim C: bind queue-nya ke topic exchange yang sama dengan pola `#`** (cocok dengan semua routing key). Producer hanya mengirim ke satu exchange, jadi semua tim cukup bind ke exchange itu. Fanout exchange terpisah hanya bisa dipakai bila producer juga mengirim ke fanout itu (setiap pesan dikirim dua kali).
- **Direct exchange** hanya untuk key yang persis sama, sehingga tidak cocok untuk pola.

6.10 P10. Ketahanan microservice: fan-out dan service lambat

Soal

Halaman utama memanggil 8 service. Saat service rekomendasi lambat, seluruh halaman ikut lambat dan akhirnya server kehabisan thread. Jelaskan penyebab dan solusinya.

Jawaban

- **Penyebab:** fan-out membuat satu request bergantung pada banyak service. Panggilan ke service lambat menahan thread, dan pada beban tinggi semua thread habis menunggu ("service hosed"). Kegagalan satu service merambat (cascading failure).
- **Solusi:**
  - **timeout** dan **circuit breaker**: service yang terus gagal tidak dipanggil dulu, langsung fallback (mis. rekomendasi default);
  - **bulkhead/thread pool terpisah** per dependency (pola Hystrix), agar satu dependency tidak menghabiskan semua thread;
  - **cache hasil panggilan** dengan TTL;
  - **panggilan paralel/asynchronous** dan **batching**;
  - meneruskan data yang sering dibutuhkan lewat header agar service hub tidak dipanggil berulang.
- **Prinsip:** *design for failure*; uji ketahanannya, mis. dengan chaos engineering.

6.11 P11. Migrasi monolith ke microservice

Soal

Aplikasi monolith besar ingin dipindah ke microservice tanpa menghentikan layanan. Jelaskan strategi dekomposisi dan komponen pendukungnya.

Jawaban

- **Strangler pattern:** fitur baru atau fitur yang dipisah dibangun sebagai microservice. API gateway mengarahkan route tertentu ke service baru, sedangkan sisanya tetap ke monolith, sampai monolith bisa dipensiunkan. Tidak ada big-bang rewrite.
- **Cara memecah:** per **kapabilitas bisnis** atau per **sub-domain/bounded context** (DDD); satu service satu tim.
- **Data:** idealnya database per service. Transaksi lintas service memakai **Saga**, dan query gabungan memakai **CQRS** atau aggregator.
- **Pendukung:** service registry/discovery; konfigurasi eksternal (12-factor); centralized logging, metrics, dan distributed tracing; CI/CD dan deployment terotomasi (blue-green/canary).
- **Trade-off:** kompleksitas operasional naik, sehingga lakukan bertahap dan hanya bila organisasi cukup besar.

7 Ringkasan Inti

Bawa ke ruang ujian (di kepala)

- **Arah soal:** dari hafalan (2022) ke perbandingan (2023) ke **kasus dan desain** (2025). Siapkan satu desain arsitektur, satu troubleshooting, dan beberapa trade-off.
- **Desain arsitektur:** requirement menjadi kebutuhan kualitas → hitung beban dari angka di soal → pisahkan jalur kritis dan jalur asynchronous → pilih komponen (gateway, load balancer, service stateless, cache, queue, circuit breaker, token auth) → gambar → skenario gagal dan trade-off.
- **Overload:** pastikan penyebab → lindungi (rate limit, antrean, circuit breaker) → kurangi beban (cache, index, pooling, queue + worker, scale out stateless) → ubah arsitektur (pisah read/write, partisi) → sebut trade-off.
- **Konsep yang sering keluar:**
  - REST (resource, HTTP, populer untuk publik) vs RPC/gRPC (fungsi, IDL, biner, HTTP/2, internal) vs GraphQL (query fleksibel, satu endpoint);
  - CQRS (pisah baca/tulis) vs Saga (transaksi lintas service + kompensasi);
  - stateless mudah di-scale, stateful butuh affinity/partisi/replikasi;
  - 12-factor build/release/run (artefak tetap, config terpisah, rollback);
  - layered (monolith berlapis) vs microservice.
- **Infrastruktur:** 5 model I/O (hanya async yang tidak blok di kedua fase); event-driven untuk banyak koneksi; middleware (RPC, message-oriented, application server, database access); API gateway (luar ke dalam) vs service mesh (antar service, sidecar); pilih message queue berdasarkan pola, volume/retensi, jaminan pengiriman, urutan, dan lingkungan.