1. (LO 3 – 30 Points) Rancang sebuah aplikasi web berbasis cloud-native sesuai topik pilihan Anda:
    a. (10 Points) Jelaskan arsitektur sistem yang Anda rancang, minimal mencakup:
        - Frontend, aplikasi dibangun menggunakan teknologi web dasar: HTML, CSS, dan JavaScript murni (Vanilla JS), tanpa ketergantungan pada framework pihak ketiga. Antarmuka pengguna ini disajikan melalui server web Nginx yang ringan dan efisien berbasis Alpine Linux, dikemas dalam container Docker terpisah yang di-expose ke port 8085. Fungsi utamanya adalah sebagai lapisan client-side yang memungkinkan pengguna berinteraksi dengan sistem melalui browser. Untuk mengakses data dan menjalankan operasi seperti manajemen layanan, pelanggan, dan kendaraan, frontend melakukan komunikasi asinkron ke Backend API menggunakan Fetch API atau AJAX.
        - Backend API, aplikasi dibangun menggunakan Python Flask sebagai framework RESTful API server. Container backend berjalan di port 5000 dan menyediakan endpoint-endpoint untuk operasi CRUD (Create, Read, Update, Delete) pada entitas utama: Customers, Vehicles, dan Services. Interaksi dengan database dilakukan secara aman dan terstruktur melalui SQLAlchemy sebagai ORM. Selain itu, backend juga dilengkapi dengan instrumentasi observability menggunakan library prometheus_client, yang mengekspos metrik kustom seperti jumlah permintaan HTTP dan latensi respons melalui endpoint /metrics, memungkinkan integrasi langsung dengan sistem monitoring eksternal.
        - Database, Database utama sistem menggunakan PostgreSQL versi 15, dijalankan dalam container Docker berbasis image postgres:15-alpine untuk efisiensi sumber daya. PostgreSQL bertindak sebagai penyimpanan data relasional terpusat untuk seluruh entitas aplikasi. Untuk memastikan persistensi data—sehingga informasi tetap utuh meskipun container di-restart atau dihapus—digunakan Docker Volume bernama postgres_data. Akses ke database dibatasi hanya untuk container backend melalui jaringan internal Docker bernama workshop-net, meningkatkan keamanan dengan mencegah akses langsung dari luar lingkungan aplikasi.
        - Monitoring/Observability, Sistem observability diimplementasikan menggunakan Prometheus Stack yang terdiri dari tiga komponen utama: Prometheus, Node Exporter, dan Grafana. Prometheus bertindak sebagai server pengumpul metrik, yang secara berkala (scrape) mengambil data dari dua sumber: endpoint /metrics pada Backend API (untuk metrik aplikasi seperti jumlah request dan latensi database) dan Node Exporter (untuk metrik infrastruktur seperti CPU, memori, dan penggunaan disk). Node Exporter dijalankan dalam container terpisah untuk memantau

kesehatan sistem host/container. Semua data ini divisualisasikan secara real-time melalui Grafana, yang dijalankan di port 3001 dan dikonfigurasi menggunakan Prometheus sebagai data source. Seluruh komponen monitoring ini diintegrasikan dalam satu jaringan Docker (workshop-net) dan dikelola secara terpadu melalui Docker Compose untuk deployment yang sederhana dan aman.

b. (10 Points) Jelaskan bagaimana komunikasi antar komponen dilakukan dan bagaimana sistem di-deploy (containerisation dan/atau orchestration).

Komunikasi dalam sistem ini terjadi antara komponen client-side (browser pengguna) dan server-side (container Docker). Pengguna mengakses antarmuka melalui http://localhost:8085, yang dilayani oleh container Nginx yang mengirimkan file statis (HTML, CSS, JS). JavaScript di browser kemudian berkomunikasi langsung ke Backend API di http://localhost:5000/api/... menggunakan Fetch/AJAX—karena permintaan berasal dari luar container, port 5000 harus dipetakan ke host. Di sisi server, aplikasi Flask terhubung ke PostgreSQL menggunakan hostname db, yang diterjemahkan oleh DNS internal Docker ke alamat IP container database dalam jaringan terisolasi workshop-net. Prometheus juga berjalan dalam jaringan ini dan secara berkala melakukan *scrape* metrik dari http://backend:5000/metrics dan http://node-exporter:9100/metrics.

Seluruh komponen dikemas dalam container Docker untuk menjamin konsistensi lingkungan: frontend menggunakan nginx:alpine, backend dibangun dari image Python, database memakai postgres:15-alpine, dan monitoring menggunakan image resmi Prometheus, Grafana, serta Node Exporter. Orkestrasi dikelola oleh docker-compose.yml, yang menangani dependensi layanan, membuat jaringan internal workshop-net untuk komunikasi aman, serta mengatur persistensi data melalui volume postgres_data dan bind mounts untuk konfigurasi eksternal seperti prometheus.yml dan dashboard Grafana. Hanya port yang diperlukan—8085 (frontend), 5000 (backend API), dan 3001 (Grafana)—yang diekspos ke host, menjaga keamanan dan kesederhanaan deployment.

c. (10 Points) Tentukan minimal 8 metrik yang menurut Anda paling penting untuk memahami perilaku sistem (performansi & reliability).

- Jelaskan alasan pemilihan setiap metrik.
    - **Requests per Second (RPS)**
      Menunjukkan beban traffic sistem. Sehat jika stabil atau sesuai pola penggunaan normal; bermasalah jika ada lonjakan tiba-tiba (DDoS/viral) atau drop ke nol (downtime/jaringan gagal).
    - **p95 Response Time**
      Mengukur pengalaman pengguna: 95% request lebih cepat dari nilai ini. Sehat jika <200 ms (ideal <50 ms); spike tinggi

mengindikasikan masalah seperti locking DB, kode tidak efisien, atau kekurangan resource.

- **Error Rate (4xx/5xx)**
  Mengukur keandalan layanan. Sehat di 0–0.1%; >1% perlu investigasi, dan >5% (terutama 5xx) menandakan insiden kritis seperti bug atau database down.
- **Database Latency**
  Mengidentifikasi bottleneck di lapisan database. Sehat jika <10 ms; kenaikan bertahap mengindikasikan kurangnya indexing atau pertumbuhan data tanpa optimasi.

- Jelaskan kondisi nilai metrik yang menunjukkan sistem sehat dan bermasalah.
  - **Backend CPU & Memory Usage**
    Memantau utilisasi resource container. Sehat jika CPU 20–60% dan memori stabil; bermasalah jika CPU 100% (throttling) atau memori terus naik (memory leak), yang bisa picu OOM kill.
  - **Disk Space Available**
    Mencegah crash akibat kehabisan ruang disk. Sehat jika tersedia >20%; <10% perlu peringatan, dan 0-byte menyebabkan sistem gagal (misal database masuk mode read-only).
  - **Network I/O**
    Memantau lalu lintas jaringan container. Sehat jika sesuai pola RPS; anomali seperti traffic tinggi dengan RPS rendah bisa mengindikasikan transfer file tak wajar atau scraping.

2. (LO 4 – 50 Points) Pada soal ini Anda wajib menggunakan simulasi beban (load simulation) menggunakan perintah di command line.
   a. (10 Points) Skenario Simulasi.

      Buat minimal dua skenario simulasi:

      - skenario beban normal,
      - skenario beban tinggi,

      Untuk setiap skenario:

      - tuliskan perintah simulasi yang digunakan,
      - jelaskan jumlah user konkuren / request / durasi,
      - jelaskan endpoint yang diuji.

Skenario Simulasi Beban Normal:

Simulasi beban normal dengan mengirim 100 request GET ke **endpoint http://localhost:5000/api/vehicles**, masing-masing dipisahkan jeda 1 detik, sehingga menghasilkan rata-rata **1 request per detik (1 RPS)** selama sekitar 100 detik. Ini merepresentasikan pola penggunaan harian yang ringan dan realistis. Dari hasil monitoring di dashboard Grafana, terlihat bahwa sistem merespons dengan sangat baik: **RPS stabil**, **p95 response time tetap di bawah 10 ms**, **error rate 0%**, **latensi database rendah (1–3 ms)**, serta **CPU dan memory backend tidak mengalami lonjakan berlebihan**. Semua indikator menunjukkan bahwa aplikasi berada dalam kondisi sehat dan mampu menangani beban normal tanpa masalah performa atau keandalan.

Perintah yang dijalankan:

```
1..60 | % {
  Invoke-RestMethod -Uri "http://localhost:5000/api/vehicles" -Method Get;
  Write-Host "Request $_ sent...";
  Start-Sleep -Seconds 1
}
```

Skenario Simulasi Beban Tinggi:

Skenario beban tinggi ini dijalankan dengan 3 terminal bersamaan (3 user konkuren), masing-masing mengirim 500 iterasi POST dan GET ke **endpoint /api/customers**, total **3.000 request tanpa jeda** — menghasilkan lonjakan RPS hingga >50 req/s, latency naik signifikan (terutama p95 dan DB latency), serta CPU backend melonjak tajam. Grafik menunjukkan sistem masih stabil tanpa error (error rate ~0%), meski database agak tertekan — membuktikan kapasitas sistem dalam menangani beban berat secara sementara.

Perintah yang dijalankan:

```
1..500 | % {
  $body = @{ name = "StressTestUser"; email = "stress@test.com" } | ConvertTo-Json
  try {
    Invoke-RestMethod -Uri "http://localhost:5000/api/customers" -Method Post -Body $body -
ContentType "application/json"
    Invoke-RestMethod -Uri "http://localhost:5000/api/customers" -Method Get
  } catch { Write-Host "Error" }
}
```

b. (20 Points) Hasil Simulasi:
Berdasarkan simulasi tersebut:

- buat tabel hasil simulasi (boleh menggunakan angka realistis/simulasi),
- Sertakan minimal metrik:
  - Request per second (RPS),
  - p95 response time,
  - error rate (4xx/5xx),
  - CPU backend,
  - latency database.

Jelaskan metrik tersebut diperoleh dari panel apa di Prometheus/**Grafana**?

Tabel Simulasi Beban Normal:

Metrik	**Nilai Simulasi**	**Keterangan**
Request per Second (RPS)	1.0 req/s (rata-rata)	Stabil, sesuai ekspektasi beban ringan
P95 Response Time	6.2 ms	Sangat cepat — tidak ada tekanan pada sistem
Error Rate (4xx/5xx)	0.0%	Semua request sukses — sistem sehat
Backend CPU Usage	25 ms (peak)	Ringan, tidak ada lonjakan signifikan
Database Latency	1.8 ms	Optimal — query dieksekusi cepat tanpa delay

Tabel Simulasi Beban Tinggi:

Metrik	**Nilai Simulasi**	**Keterangan**
Request per Second (RPS)	58 req/s (peak)	Puncak saat semua thread aktif bersamaan
P95 Response Time	17 ms	Naik dari normal (<5ms) karena tekanan DB & CPU
Error Rate (4xx/5xx)	~50.2% (puncak)	Terjadi error massal saat puncak beban —

		kemungkinan database timeout atau connection pool habis
Backend CPU Usage	420 ms (peak)	Lonjakan tajam selama simulasi berlangsung
Database Latency	3.8 ms (rata-rata puncak)	Meningkat dari baseline ~1.5 ms, menunjukkan bottleneck I/O

Panel diambil dari **Grafana**

c. (20 Points) Analisis dan Diagnosa.

- Analisis minimal 2 akar masalah performa/reliability berdasarkan hasil simulasi.
  - **Database Connection Exhaustion**
    PostgreSQL memiliki batas koneksi (default 100). Saat beban tinggi, Flask-SQLAlchemy mencoba membuka koneksi baru untuk tiap request. Jika semua slot terisi, request baru ditolak atau timeout, menyebabkan error 500 karena aplikasi gagal berkomunikasi dengan database.
  - **Worker Starvation**
    Flask dengan worker sinkron memproses request satu per satu. Jika operasi database (misalnya INSERT ke tabel Customers) melambat karena lock contention, worker akan terblokir. Request baru menumpuk di antrian TCP; jika melebihi kapasitas atau timeout, koneksi diputus dan menghasilkan error 50x.
- Sebutkan minimal 2 metrik tambahan yang diperlukan untuk analisis lebih akurat dan alasannya.
  - **Active Database Connections (Pool Usage)**
    Metrik ini (misal pg_stat_activity_count) menunjukkan jumlah koneksi aktif ke PostgreSQL. Jika nilainya flatline di angka 100—batas max_connections—saat insiden terjadi, maka *connection exhaustion* terkonfirmasi sebagai akar masalah.
  - **Container CPU Throttling**
    Metrik seperti container_cpu_cfs_throttled_seconds_total mengungkap apakah container dibatasi oleh Docker karena melebihi kuota CPU. Meski CPU usage tampak normal, throttling tinggi menyebabkan latency melonjak karena proses sementara "dibekukan" oleh sistem operasi.
- Jelaskan alur incident response yang Anda lakukan ketika p95 latency dan error rate meningkat tajam.

Dalam 0–5 menit, insiden terdeteksi melalui alert dari Grafana/Slack; dashboard menunjukkan spike RPS yang tidak wajar, mengindikasikan load test atau serangan. Pada 5–15 menit, mitigasi dilakukan dengan restart container backend untuk membersihkan antrian dan koneksi DB macet, atau scale up jika trafik valid. Di fase analisis (15–60 menit), log backend diperiksa untuk error seperti "Connection refused", lalu diperbaiki dengan menerapkan connection pooling (misal PgBouncer) atau caching Redis. Sehari setelah insiden, laporan post-mortem dibuat berisi akar masalah, temuan, dan langkah pencegahan—seperti menambahkan rate limiting di Nginx—agar kejadian serupa tidak terulang.

3. (LO 5 – 20 Points) Berdasarkan hasil analisis pada Soal 2:
    a. (10 Points) Evaluasi keputusan arsitektur dan operasional sistem Anda:
        - Sebutkan 2 keputusan yang sudah tepat
            - **Arsitektur Terpisah (Decoupled Frontend & Backend)** Memisahkan frontend (Nginx + file statis) dan backend (Flask API) ke container berbeda memungkinkan scaling independen, penggantian teknologi frontend tanpa mengganggu backend selama API konsisten, serta meningkatkan keamanan dengan memisahkan lapisan presentasi dari logika data.
            - **Observabilitas "Pull-Based" dengan Prometheus** Menggunakan Prometheus untuk *scrape* metrik dari endpoint /metrics (bukan push) membuat sistem lebih andal—jika monitoring down, aplikasi tetap berjalan normal. Pendekatan ini juga mengikuti standar cloud-native, memudahkan integrasi dengan ekosistem seperti Kubernetes dan Grafana di masa depan.
    b. (10 Points) Usulkan minimal 3 improvement jangka menengah/panjang, misalnya
        - autoscaling,
        - caching,
        - optimasi database,
        - optimasi backend,
        - efisiensi resource,
        - (opsional) optimasi atau penerapan machine learning.

        Untuk setiap improvement:
        - Jelaskan dampaknya terhadap performa, reliability, cost, dan sustainability,
        - Sebutkan metrik untuk mengukur keberhasilan.

- **Implementasi Caching Layer (Redis)**
  Menambahkan Redis untuk menyimpan data yang sering dibaca (seperti Vehicles atau Services) meningkatkan kecepatan read hingga sub-milidetik, mengurangi beban database, dan mencegah kegagalan saat trafik tinggi. Keberhasilannya diukur dari cache hit ratio >80% dan penurunan CPU database 30–50%.
- **Autoscaling & Orchestration (Kubernetes HPA)**
  Migrasi ke Kubernetes dengan Horizontal Pod Autoscaler memungkinkan backend otomatis menyesuaikan jumlah pod sesuai beban—menjaga performa stabil, menghemat biaya saat sepi, dan meningkatkan keandalan lewat self-healing. Metrik keberhasilan: antrian request tetap rendah dan utilisasi CPU pod stabil di sekitar target (misal 60%).
- **Database Connection Pooling & Read Replicas**
  Menggunakan PgBouncer menghilangkan overhead pembuatan koneksi baru dan mencegah error “too many connections”, sementara read replica membagi beban query. Hasilnya: throughput transaksi naik dan waktu tunggu koneksi mendekati nol—meningkatkan performa dan stabilitas sistem.