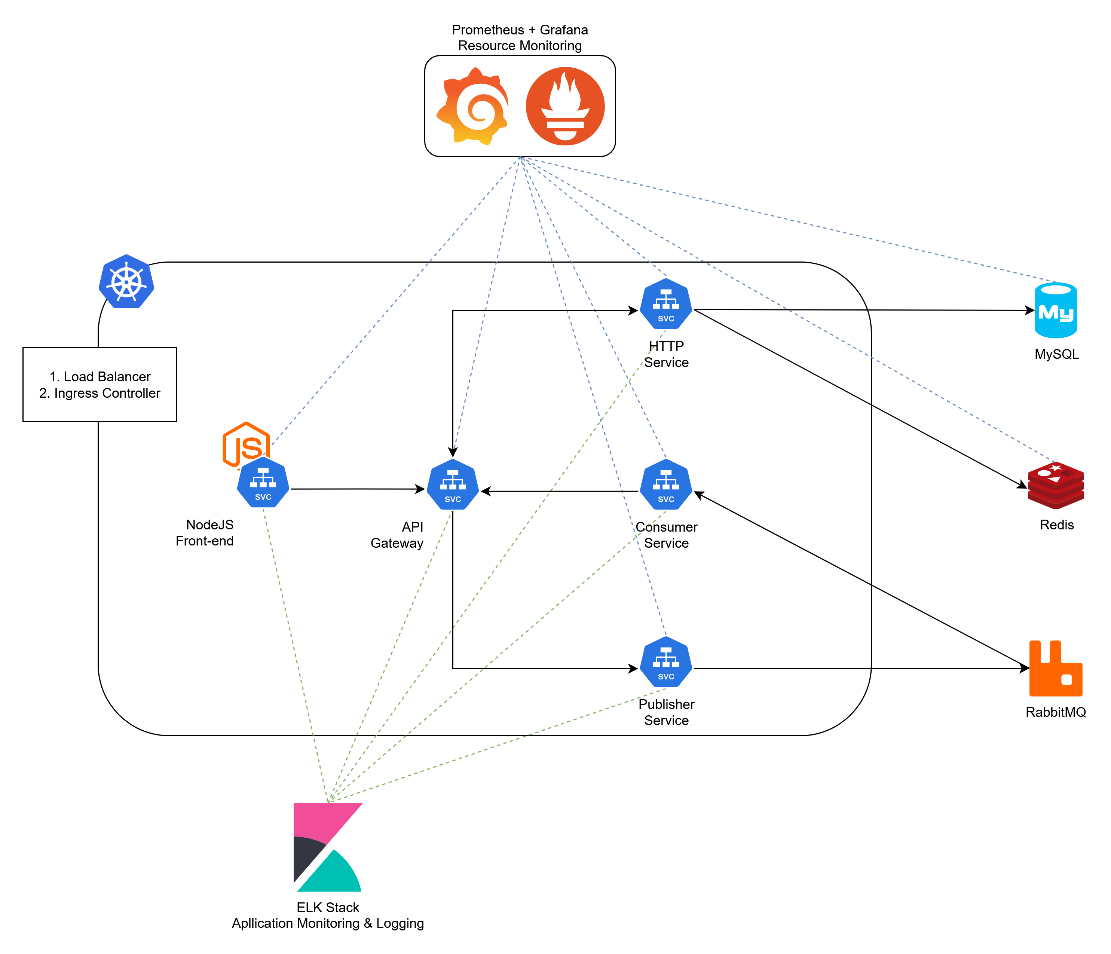
**TASK 1**

1. 
2. **Simplified Flow**:

User request masuk melalui Kubernetes LoadBalancer dan Ingress Controller, kemudian diarahkan ke Node.js Front-end Service. Front-end akan meneruskan request ke API Gateway Service, yang berfungsi sebagai entry point terpusat ke beberapa backend microservices: HTTP Service, Publisher Service, dan Consumer Service.

Setiap service dijalankan sebagai Deployment di Kubernetes untuk mendukung scaling dan menghindari single point of failure. Selain itu, Horizontal Pod Autoscaler (HPA) dapat digunakan untuk menyesuaikan jumlah Pod sesuai beban. Untuk konfigurasi service disimpan dalam ConfigMaps, sementara data sensitif seperti kredensial database, Redis, dan RabbitMQ disimpan sebagai Secrets.

Untuk arsitektur Event-Driven, service berkomunikasi secara asynchronous melalui RabbitMQ, publisher-service dan consumer-service bekerja berdasarkan trigger yang dipicu oleh http-service atau kondisi tertentu. Di sisi observability, Prometheus + Grafana digunakan untuk monitoring resource tiap node, sedangkan ELK Stack digunakan untuk analisis performa aplikasi serta system log dari setiap service.

**Detail Flow**:

Ketika request dari user eksternal masuk ke cluster Kubernetes, trafik pertama kali diarahkan melalui LoadBalancer Service atau NodePort tergantung environment. Load balancer ini kemudian mengirimkan request ke Ingress Controller (misalnya Nginx, Traefik, atau AWS ALB Controller) yang berfungsi membaca resource Ingress dan menentukan routing berdasarkan host atau path yang diminta. Dari situ, request diteruskan ke Service frontend yang sudah terdaftar di cluster. Service frontend memiliki DNS internal yang diterjemahkan oleh CoreDNS menjadi ClusterIP, sementara kube-proxy di setiap node menyiapkan aturan iptables atau ipvs untuk melakukan load balancing ke salah satu Pod frontend yang sehat. Endpoint Controller memastikan hanya Pod dengan status Ready yang menjadi target routing.

Setelah sampai di Pod frontend (Node.js), aplikasi frontend melayani static content atau meneruskan request ke backend melalui API Gateway. API Gateway ini berperan sebagai pintu masuk utama backend dan meneruskan request sesuai jenisnya. Untuk operasi synchronous seperti CRUD data, request diarahkan ke HTTP Service yang berinteraksi langsung dengan MySQL sebagai database utama dan Redis sebagai cache atau session store. Sementara untuk proses asynchronous atau event-driven, API Gateway meneruskan request ke Publisher Service yang akan mengirim event ke RabbitMQ. Event yang masuk ke antrean RabbitMQ diproses oleh Consumer Service, sehingga workload asynchronous dapat dipisahkan dari request synchronous.

Seluruh konfigurasi non-sensitif seperti endpoint aplikasi dikelola dengan ConfigMaps, sedangkan kredensial sensitif seperti password database, Redis, dan RabbitMQ disimpan menggunakan Secrets. Setiap service dijalankan sebagai Deployment yang mengontrol ReplicaSet untuk memastikan jumlah Pod sesuai definisi, dengan strategi rolling update agar aplikasi tetap tersedia saat upgrade. Skalabilitas dijaga oleh Horizontal Pod Autoscaler (HPA) yang dapat menambah atau mengurangi jumlah Pod secara otomatis, baik berdasarkan resource usage seperti CPU dan memori, maupun metrik kustom seperti panjang antrean RabbitMQ. PodDisruptionBudget (PDB) juga digunakan untuk mencegah semua Pod mati bersamaan saat ada maintenance atau upgrade node.

Untuk observabilitas, Prometheus mengumpulkan metrik dari node, Pod, Service, serta komponen eksternal seperti MySQL, Redis, dan RabbitMQ. Semua metrik tersebut divisualisasikan dalam Grafana sehingga tim dapat memantau performa dan resource usage secara real time. Di sisi logging, agent seperti Fluentd, Fluentbit, atau Filebeat dijalankan sebagai DaemonSet di setiap node untuk mengumpulkan log container dari stdout/stderr. Log tersebut kemudian dikirim ke Elasticsearch (melalui Logstash atau langsung) dan divisualisasikan dengan Kibana, sehingga tim dapat melakukan analisis error, troubleshooting, maupun pelacakan request lintas service secara lebih mendalam.

1. **High availability** berarti bahwa suatu sistem IT, komponen, atau aplikasi dapat beroperasi pada tingkat tinggi secara terus-menerus tanpa intervensi selama periode waktu tertentu. Infrastruktur high-availability dikonfigurasi untuk memberikan performa yang berkualitas serta mampu menangani berbagai beban dan kegagalan dengan downtime yang minimal atau bahkan nol.

**Auto scaling** adalah teknik dalam cloud computing untuk mengalokasikan sumber daya komputasi secara dinamis. Bergantung pada beban yang diterima oleh **server farm** atau **server pool**, jumlah server yang aktif biasanya akan berubah secara otomatis sesuai dengan fluktuasi kebutuhan pengguna

**Software housekeeping** adalah pendekatan holistik untuk menjaga kesehatan, keterbacaan, dan efisiensi proyek perangkat lunak. Dengan perencanaan yang cermat, menghilangkan elemen yang tidak diperlukan, mengikuti konvensi penamaan yang konsisten, menjaga integritas struktur, mendokumentasikan secara menyeluruh, dan membersihkan komponen yang tidak terpakai, developer dapat menciptakan solusi perangkat lunak yang tahan lama. Sama seperti rumah yang bersih dan tertata rapi dapat meningkatkan kesejahteraan, **codebase** yang terawat dengan baik akan mendukung pengembangan yang lebih lancar, pemeliharaan yang lebih mudah, dan *Software Development Life Cycle* yang lebih berkelanjutan.

**Monitoring** adalah proses *mengumpulkan*, *memproses*, *mengagregasi*, dan *menampilkan* data kuantitatif secara real-time tentang sistem teknologi informasi. Data ini dapat mencakup jumlah dan jenis *query*, jumlah dan jenis *error*, waktu pemrosesan, serta *lifetime* dari server.

**TASK 5**

1. **Root Cause Analysis in Production Environtment** (Lonjakan 4xx / Downtime 5xx)

• Periksa dashboard monitoring /alerting (APM, logs, tracing, metrics) untuk anomali

• Identifikasi apakah isu bersifat global atau terisolasi pada region/service tertentu

**Untuk 4xx (Lonjakan Client Errors):**

• Konfirmasi apakah disebabkan oleh perubahan aplikasi (deployment terbaru, config drift)

• Inspeksi log API Gateway/Ingress (netstat, ss, lsof, tcpdump, kubectl logs, kubectl describe ingress)

• Me-review auth/ACL/Rate-limit policy — lonjakan 401/403 bisa berasal dari expired token, IAM yang salah konfigurasi, header auth kosong, redis / memchaced overload

**Untuk 5xx (Downtime / Unreachable):**

• Periksa health checks & error budgets (aplikasi down, config drift di networking, status infra, IAM role, SLO)

• Inspeksi log service mesh / ingress controller / load balancer

• Cek infra: node OOM, saturasi DB, network partition, expired TLS/Cert, DNS propagation

• Rollback deployment terbaru jika ada korelasi

**Konfirmasi Akar Masalah:**

• Reproduksi di staging jika memungkinkan

• Validasi dengan log, metrics, dan korelasi dampak ke pengguna

2. **Menyelesaikan Tugas Besar Tepat Waktu**

• **Decompose Work**: Pecah tugas besar menjadi milestone kecil dengan deliverables yang jelas

• **Define Acceptance Criteria**: Samakan ekspektasi sejak awal dengan stakeholder atau tim terkait

• **Prioritize Risky Items Early**: Tangani dependency, unknowns, dan kebutuhan infra terlebih dahulu

• **Continuous Progress Tracking**: Gunakan agile, kanban, & pantau burn-down chart

• **Transparent Communication**: Tandai blocker sejak awal, hindari kejutan menjelang deadline

• **Automate Where Possible**: CI/CD, testing, dan monitoring untuk mengurangi beban manual

3. **VPC Subnet Strategy for Production**

Minimal yang direkomendasikan:

• Public Subnet: Untuk load balancer, NAT gateway, bastion host / SSH ke private instance

• Private Subnet: Untuk app server, container, backend service (Non-facing Public IP)

• Data Subnet: Untuk DB, cache, message queue, idealnya dengan NACL dan SG yang lebih ketat

4. **Optimasi Biaya Cloud / FinOps**

• **Right-Sizing**: Gunakan monitoring (CloudWatch, Datadog, Prometheus) untuk mendeteksi resource yang underutilized dan lakukan downsizing

• **Auto-Scaling**: Aktifkan HPA (Kubernetes ) atau ASG (EC2) dengan policy berbasis metric (CPU, queue length, custom Prometheus), bukan dengan kapasitas tetap (perlu testing).

• **Storage Lifecycle**: Gunakan tiering object storage (S3 IA/Glacier), hapus snapshot yang tidak digunakan atau pindah ke S3 Glacier / Archive.

• **Reserved/Spot Instances**:

Untuk workload yang bisa diprediksi, gunakan RIs/Savings Plans;

Untuk job non-kritis gunakan Spot.

• **Network Optimization**: Kurangi transfer data antar AZ dengan colocating service

• **Serverless Adoption**: Migrasikan cron job, batch task, atau API ringan ke Lambda/FaaS jika memungkinkan

• **Initiatives**: Lakukan review biaya secara berkala (FinOps sprint), terapkan tagging untuk chargeback/showback

5. Praktik Terbaik Keamanan (Infra Public Cloud)

Access Control:

• Terapkan IAM least privilege (berbasis role, kredensial jangka pendek, tanpa key jangka panjang)

• Gunakan SSO dengan MFA untuk engineer

Application Security:

• Gunakan WAF/Shield untuk proteksi API dan CAPTCHA

• Aktifkan TLS di semua tempat (ACM certs, TLS v1.2+)

Database Security:

• Hanya gunakan private subnet, whitelist Security Group

• Enkripsi data saat disimpan (KMS) dan saat ditransmisikan (TLS)

• Rotasi kredensial DB menggunakan Secret Manager atau Vault

Credential Management:

• Jangan hardcode secret di kode atau ENV → gunakan Secret Manager/SSM Parameter Store

• Rotasi access key secara berkala

Monitoring & Audit:

• Aktifkan CloudTrail, GuardDuty, SecurityHub

• Logging terpusat dengan integrasi SIEM

Infra Hardening:

• Terapkan CIS benchmark

• Wajibkan patching, gunakan pipeline AMI

Zero Trust Practices:

• Gunakan identity-aware proxy, microsegmentation, dan verifikasi berkelanjutan