Service Design and Decomposition
Each service must be designed around a single business capability. If the service needs to expose functionality externally, use REST APIs. If it communicates primarily with other internal services, use gRPC for better performance. Services should remain stateless, with session data stored externally. If services must coordinate to complete a workflow, evaluate the transaction model. If strong consistency is required, apply the Saga pattern. Otherwise, use asynchronous events and compensating transactions. Services that require eventual consistency must include retry logic and failure handling. If the design involves stateful behavior, skip directly to Data Persistence and Management to ensure persistence requirements are defined early. Otherwise, continue to Containerization.

Containerization
All services must be containerized using Docker. Use multi-stage builds to separate build-time and runtime dependencies. If container images will be deployed to secure environments, apply vulnerability scanning during the CI process. Never include secrets or configuration values in the container image. If your container needs access to runtime secrets, ensure you're integrating with a secrets manager. If you are containerizing services that have runtime-sidecar dependencies (e.g., for service mesh or observability), include them now, or skip directly to Service Discovery and Routing to configure routing accordingly. If containers are built with runtime-based configuration, go directly to Service Discovery and Routing. Otherwise, continue to Orchestration with Kubernetes.

Orchestration with Kubernetes
Each service must be deployed as a Kubernetes Deployment. Use ConfigMaps for configuration and Secrets for sensitive values. If the service requires persistent volumes or must retain state, use StatefulSets instead of Deployments. In this case, before continuing, go to Data Persistence and Management to prepare persistent data handling. All services must define liveness and readiness probes. If the service is stateless, apply autoscaling policies using the Horizontal Pod Autoscaler. If it is stateful, scaling must be manually configured and tested for consistency. After orchestration is complete, proceed to Service Discovery and Routing.

Service Discovery and Routing
Services must register and resolve via Kubernetes DNS. If your system requires advanced traffic routing, retries, or circuit breakers, implement a service mesh like Istio. If Istio is installed, define VirtualServices for routing and DestinationRules for load balancing. If a service mesh is not used, configure ingress and internal routing via kube-proxy or an Ingress controller. If the service will be exposed externally, continue to API Gateway and External Access. If the service is internal-only, skip to Observability to begin configuring logging and monitoring.

API Gateway and External Access
Public-facing services must be routed through an API gateway such as Kong or Ambassador. Configure routes with JWT-based authentication and enable rate limiting. If the environment requires integration with multiple identity providers, federate access using OAuth2 and OpenID Connect. If request routing is tenant-specific, define routing rules by header or path. If this service requires secure user data access, proceed to Security before exposing the service. Once gateway routing is configured, continue to Observability.

Observability
All services must emit structured logs and metrics. Deploy Fluent Bit or Fluentd to capture logs and

forward them to Elasticsearch or Loki. Prometheus must collect metrics from services exposing a /metrics endpoint. Set up Alertmanager to monitor error rate, latency, and throughput. If tracing is required, inject OpenTelemetry into services and export data to Jaeger. Dashboards must be created in Grafana. If services are mission-critical, observability setup must be completed before implementing CI/CD and Deployment Strategies, as monitoring will control deployment rollbacks. Once observability is in place, continue to Security.

Security
All internal traffic must be encrypted using mutual TLS. If a service mesh is used, enable mesh-wide mTLS policies. Otherwise, enforce TLS termination at ingress and configure TLS downstream using application-level encryption or sidecar proxies. Role-Based Access Control must be applied in Kubernetes, and each namespace should define strict NetworkPolicies to control service-to-service communication. All container images must pass vulnerability scanning, and sensitive data must be encrypted in transit and at rest. If any service manages personal data or secrets, validate whether it requires data protection rules, and if so, return to Data Persistence and Management to configure secure persistence. After security enforcement is in place, continue to CI/CD and Deployment Strategies.

CI/CD and Deployment Strategies
The CI pipeline must validate code, run tests, and scan for vulnerabilities. The CD pipeline must support rolling updates by default. For customer-facing services, implement canary deployments or blue/green rollouts. Canary releases must integrate with observability to control rollback triggers. If feature flags are used, integrate a central feature flag system. If one is not available, return to Service Design and Decomposition to review how toggles are implemented in code. All rollback logic must be automated. Once deployment automation is verified, continue to Resilience and Failover.

Resilience and Failover
Services must be able to detect failure and isolate it. Implement circuit breakers to stop cascading failures. For messaging services, configure dead-letter queues and define idempotent retry logic. If downstream services are unavailable, the system must fall back to safe degraded modes, such as serving cached data or skipping non-essential operations. If traffic rerouting is required, return to Service Discovery and Routing and update routing logic in the service mesh or ingress. Once resilience policies are in place, continue to Data Persistence and Management.

Data Persistence and Management
All persistent data must be stored in secure, reliable databases. Use PostgreSQL or MySQL for transactional workloads, and Cassandra or DynamoDB for distributed write-heavy systems. Redis should be used for ephemeral or cached data. Databases must support automated backups and point-in-time recovery. If deploying across regions, databases must support geo-replication. Ensure encryption at rest is enabled. If any service handles personal or regulated data, return to Security to validate encryption and retention policies. After persistence strategy is complete, continue to Disaster Recovery and Multi-Region Deployment.

Disaster Recovery and Multi-Region Deployment
In multi-region deployments, use cloud-native load balancers with health checks and failover. Each region must operate autonomously with local instances of critical services like authentication, configuration, and observability. Route global traffic based on latency or availability. Ensure that

persistent storage is replicated across regions or backed up regularly. Validate recovery plans through quarterly simulations. If failover recovery depends on Kubernetes orchestration, return to Orchestration with Kubernetes to ensure the control plane is configured for high availability.