Job Scope — Backend Dev, Node.js ecosystem, Express, MongoDB, Redis, Kafka, Docker, Kubernetes, AWS, Terraform.
Application — Company's first app. Must handle 100K concurrent users, 50K requests/sec, 5 million rows in 30 seconds. Also internal as well.

Deployment:
  Traditional — Single server, manual deploy.
    Pros: Simple (one VM, one log).
    Cons: Single point of failure (crash, reboot, etc).
  Docker Container —
    Pros: Reproducible, runs anywhere (laptop, staging, prod).
    Cons: Overhead, need orchestration (Docker alone not enough).
  Kubernetes — Pods, auto-scaling, rolling updates, self-healing.
    Cons: Complex, overkill for small teams.

Concurrency and Scalability
  Job asks about APIs, Kafka, distributed, and multi-region.
  Node.js —
    Event-loop —
      1 — Express receives GET /orders/:id request.
      2 — Route handler calls Redis first (cache).
        Cache hit → return JSON immediately.
        Cache miss → query MongoDB, then write result back to Redis.
      3 — Response sent, event loop moves to next request.
      Key insight — Single thread, but never blocks on I/O because everything is async.
    Cluster —
      Node.js runs one process per CPU core by default.
      Master process forks worker processes, each handles requests independently.
      Shared state must live in Redis or MongoDB, not in memory.
    Worker threads —
      Use for CPU-heavy work (image resizing, PDF generation, crypto).
      Don't use for I/O — event loop handles that better.
  Redis —
    Cache — Reduce DB load, sub-millisecond response times.
    Session store — Stateless servers, sessions survive restarts.
    Rate limiting — Sliding window counter per IP or API key.
    Pub/sub — Lightweight messaging between services.
    Caveat — Not persistent by default, data can be lost on restart.
  Kafka —
    Producer — Service publishes events to a topic.
    Consumer group — Multiple consumers share partitions, each processes a subset.
    Offset — Consumer tracks position, can replay from any point.
    Use cases — Order events, notifications, analytics, cache invalidation across services.
    Caveat — Eventually consistent, ordering only within a partition.
  Multi-region —
    Latency — Cross-region DB reads add 100–200ms, unacceptable for user-facing APIs.
    Strategy — Read replicas per region, writes go to primary region.
    Cache — Each region has its own Redis, invalidation must propagate across regions.
    Trade-off — Stale reads in secondary regions vs. slow writes everywhere.

Kubernetes — Pods, Deployments, Services, Ingress, HPA, rolling updates, health checks.
  Pod — Smallest unit, one or more containers, shared network and storage.
  Deployment — Desired state, K8s keeps pods matching it, supports rolling updates and rollbacks.
  Service — Stable network endpoint for a set of pods, load balances between them.
  Ingress — Routes external HTTP traffic to services, handles TLS termination.
  HPA — Horizontal Pod Autoscaler, scales pods based on CPU, memory, or custom metrics.
  Health checks — Liveness (restart if dead), readiness (stop traffic if not ready).
  Caveat — Steep learning curve, overkill for small teams, requires dedicated ownership.

Terraform — Infrastructure as code, declarative, state file, modules, plan before apply.
  Provider — Connects to AWS, GCP, Azure, or any API.
  Resource — Defines a single infrastructure object (VPC, EC2, RDS, S3, etc).
  State file — Tracks what exists, must be remote (S3 + DynamoDB) for team use.
  Module — Reusable group of resources, like a function.
  Workflow — init → plan → review → apply → commit.
  Caveat — Destroy can be dangerous, always review plan output carefully.

Observability — Logs, metrics, traces, alerts, dashboards, SLO/SLI/SLA.
  Logs — Structured JSON, ship to centralized store (ELK, Loki, CloudWatch).
  Metrics — Prometheus + Grafana, RED method (Rate, Errors, Duration).
  Traces — Distributed tracing (OpenTelemetry, Jaeger), follow a request across services.
  Alerts — Page on symptoms, not causes, avoid alert fatigue.
  SLO — Service Level Objective, internal target (e.g. 99.9% uptime).
  SLI — Service Level Indicator, measured metric (e.g. p99 latency).
  SLA — Service Level Agreement, contract with customers, usually looser than SLO.
  Caveat — Observability is useless without someone reading the dashboards and acting on them.

Security — Auth, JWT, OAuth, rate limiting, input validation, secrets management, CVE scanning.
  Auth — Session cookies vs JWT, OAuth for third-party login.
  JWT — Stateless, but cannot revoke easily, keep expiry short (15 min).
  Rate limiting — Sliding window per IP or API key, Redis-backed.
  Input validation — Never trust client data, validate at the boundary.
  Secrets — Never in code or Dockerfile, use vault or cloud secrets manager.
  Dependabot — Automated CVE scanning and PR for vulnerable dependencies.
  Caveat — Security is a process, not a feature, one CVE can ship to production.

Data — Schema design, indexes, transactions, replication, sharding, backup and restore.
  Schema design —
    Embed — Data accessed together, bounded size, one-to-few relationship.
    Reference — Data grows unbounded, many-to-many, or accessed independently.
    Rule — Optimize for the most common query, not for normalization purity.
  Indexes —
    Single field — Simple queries on one field.
    Compound — Multi-field queries, order matters (ESR rule: Equality, Sort, Range).
    Text — Full-text search, limited compared to Elasticsearch.
    Geospatial — Location queries, 2dsphere for real-world coordinates.
    Caveat — Every index slows writes and uses memory, monitor with explain().
  Transactions —
    Multi-document — Atomicity across collections, use sparingly, adds latency.
    Caveat — Prefer single-document atomicity by design, transactions are expensive.
  Replication —
    Primary-secondary — Writes to primary, reads from secondaries, eventually consistent.
    Caveat — Replication lag means stale reads, not suitable for strong consistency.
  Sharding —
    Shard key — Must match most queries, poor choice causes scatter-gather.
    Caveat — Irreversible in practice, resharding is painful, avoid until necessary.
  Backup —
    Mongodump — Logical backup, slow for large datasets.
    Snapshot — Fast, consistent, preferred for production.
    Point-in-time — Oplog replay, recover to any moment within retention window.
    Caveat — Untested backups are not backups, test restore regularly.

Stack Choice — Node.js + Express + MongoDB + Redis + Kafka + Docker + Kubernetes + AWS + Terraform.
  Why Node.js — Single language across frontend and backend, strong ecosystem, fast iteration.
  Why Express — Minimal, well-known, easy to hire for, enough structure for a small team.
  Why MongoDB — Flexible schema, fits document-shaped data, easy to start, scales horizontally.
  Why Redis — Sub-millisecond latency, multi-purpose (cache, session, rate limit, pub/sub).
  Why Kafka — Durable event log, decouples services, enables replay and audit trail.
  Why Docker — Reproducible deployments, same image from laptop to production.
  Why Kubernetes — Self-healing, rolling updates, horizontal scaling, declarative infrastructure.
  Why AWS — Most mature ecosystem, managed services reduce operational overhead.
  Why Terraform — Infrastructure as code, reviewable, reproducible, multi-cloud capable.
  Caveat — Complexity compounds, every tool adds a failure mode and a learning curve.

Trade-offs — Consistency vs availability, latency vs throughput, simplicity vs scalability, cost vs resilience.
  Consistency vs availability — Strong consistency means slower writes and possible downtime during partitions.
  Latency vs throughput — Batching improves throughput but adds latency to individual requests.
  Simplicity vs scalability — Start simple, add complexity only when the bottleneck actually appears.
  Cost vs resilience — Multi-region doubles infrastructure cost, only justified for critical services.
  Caveat — Every decision has a cost, name the trade-off explicitly, don't hide it.

Failure Modes — Cache stampede, thundering herd, cascading failure, split brain, data loss, silent corruption.
  Cache stampede — Many requests miss cache simultaneously, overload the database.
    Fix — Lock per key, stale-while-revalidate, jitter on TTL.
  Thundering herd — Many clients retry at the same time after a failure.
    Fix — Exponential backoff with jitter, circuit breaker.
  Cascading failure — Slow dependency ties up threads, upstream services time out.
    Fix — Timeouts everywhere, circuit breaker, bulkhead isolation.
  Split brain — Two nodes both think they are primary, writes diverge.
    Fix — Quorum writes, fencing tokens, leader lease with expiry.
  Data loss — Replication lag, crash before write, or bad deploy.
    Fix — Backups, point-in-time recovery, staging environment, canary deploys.
  Silent corruption — Bit rot, bad disk, or bug writes wrong data.
    Fix — Checksums, periodic integrity scans, restore from backup.
  Caveat — Most incidents are combinations of these, not single failures.

Interview Stories — Prepare 3 stories: a production incident you fixed, a system you designed from scratch, a trade-off you made and defended.
  Incident story — Symptom, root cause, fix, prevention, what changed in the runbook.
  Design story — Requirements, constraints, alternatives considered, why you chose what you chose.
  Trade-off story — Two valid options, what you gained, what you lost, when you would reverse it.
  Caveat — Specific numbers beat vague claims, "reduced p99 latency from 2s to 200ms" not "improved performance".

Mindset — Ship small, measure before optimizing, write runbooks, own your code in production.
  Ship small — Feature flags, canary deploys, rollback in minutes not hours.
  Measure first — Profile before optimizing, intuition is wrong about bottlenecks.
  Runbooks — Document how to restart, rollback, and diagnose at 3 AM.
  Own your code — Deploy, monitor, and fix your own services, no handoff to ops.
  Caveat — Ownership without burnout, on-call should be sustainable, not heroic.