System Design (without a dedicated round) — how to signal end-to-end strength

Use this to sprinkle system design maturity during coding/ML rounds: requirements, API + data model, scaling, reliability, security, ops, cost. Keep tradeoffs explicit.

1) 2–3 minute kickoff script

- Restate problem as product: users, core workflow, critical paths. Clarify scope + non-goals.
- Get numbers: DAU/MAU, read/write QPS, payload size, p95/p99 latency, availability target, data retention.
- Define success: primary metric + guardrails (latency, correctness, privacy, abuse, cost).
- Pick a baseline architecture; mention 1–2 future upgrades. Draw diagram early (clients → edge → services → data).
- Call out risks (top 3) + mitigations (rate limiting, idempotency, backpressure, fallbacks).

2) Diagram skeleton (draw fast)

Clients → **Edge** (CDN/WAF/rate limit) → **API Gateway** → **Stateless services** (auth, core, fanout, search) → **Data** (DB/cache/blob) → **Async** (queue/stream) → **workers** (compute, indexing, notifications) → **Observability** (logs/metrics/traces) + **config/experiments** + **CI/CD**.

3) APIs & data modeling checklist

• **API:** endpoints, authZ/authN, pagination, filtering/sorting, idempotency keys for writes, versioning.
• **Entities:** define IDs (UUID/ULID), ownership, relationships, cardinalities.
• **Access patterns:** list the top queries (by key, by user, by time, search). Model around them.
• **Consistency:** what must be strongly consistent vs eventual (e.g., balances vs feeds).
• **Schema evolution:** additive changes, defaults, backfill strategy.

4) Storage choices with pros/cons

- **Relational (Postgres/MySQL):** +ACID, joins, constraints, mature tooling; –harder horizontal sharding, careful with hot rows.
- **Key-Value (Redis/Dynamo-style):** +low latency, simple scaling; –limited queries/joins, modeling upfront, eventual consistency variants.
- **Wide-column (Cassandra/HBase):** +high write throughput, predictable access by partition key; –query flexibility, operational complexity, consistency tradeoffs.
- **Search index (Elasticsearch/OpenSearch):** +full-text + filtering; –eventual consistency, tuning/ops cost, reindexing.
- **Object store (S3/GCS):** +cheap durable blobs; –not for low-latency small reads without caching/CDN.
- **Time-series/log store:** +append-heavy metrics/events; –not for OLTP queries.

5) Caching & edge: pros/cons + where it fits

- **CDN:** +offload global reads, low latency; –cache invalidation, personalization limits.
- **Service cache (Redis/Memcached):** +reduce DB load, speed reads; –staleness, stampedes, eviction surprises.
- **Patterns:** cache-aside (simple), read-through (centralized), write-through (fresh but slower), write-behind (fast but risk).
- **Stampede control:** request coalescing, probabilistic early refresh, jittered TTLs, soft TTL + background refresh.

6) Scalability mechanics (what you say out loud)

- **Scale reads:** caching, replicas, denormalized read models, precompute, pagination, avoid N+1 fanout.
- **Scale writes:** sharding/partitioning by tenant/user/time; batch, async pipelines, idempotent retries.
- **Sharding:** consistent hashing; beware hot keys; add random prefix or split heavy users.
- **Replication:** leader/follower (read replicas) vs multi-leader (conflicts) vs quorum.
- **Backpressure:** bounded queues, timeouts, shedding, priority lanes for critical traffic.

7) Reliability patterns + tradeoffs (L6 signals)

- **Timeouts + retries:** +mask transient failures; –retry storms. Use exponential backoff + jitter + budgets.
- **Circuit breaker:** +protect dependencies; –needs tuning, can cause partial outages if mis-set.
- **Bulkheads:** +isolate tenants/features; –more capacity planning.
- **Graceful degradation:** +keep core up; –reduced feature quality (serve cached, skip expensive steps).
- **Idempotency:** +safe retries; –requires keys + dedupe storage/window.
- **Exactly-once myth:** aim for at-least-once + idempotent handlers; use transactional outbox where needed.

8) Queues/streams & async processing (pros/cons)

- **Message queue (SQS/RabbitMQ):** +simple work distribution, retries/DLQ; –ordering/throughput limits depending on system.
- **Log/stream (Kafka/Pulsar):** +high throughput, replay, multiple consumers; –ops complexity, ordering per partition, schema discipline.
- **DLQ:** +prevents poison-pill blocking; –needs re-drive process + monitoring.
- **Exactly-once needs:** transactional writes or idempotent consumer + dedupe key + offset management.

9) Consistency, correctness, and distributed reality

- **CAP talk-track:** under partition you pick Consistency vs Availability; choose per operation.
- **Strong consistency:** +simpler correctness; –higher latency/less availability.
- **Eventual consistency:** +availability/latency; –stale reads, anomalies; must design UX and reconciliation.
- **Common tools:** version numbers, compare-and-swap, read-repair, conflict resolution (LWW or domain-specific).
- **Transactions:** local transactions preferred; cross-service transactions are hard → sagas/compensation.

10) Security, privacy, and abuse-resilience

- **AuthN/AuthZ:** JWT/session, service-to-service mTLS; least privilege; audit logs.
- **PII:** minimize collection, encrypt at rest/in transit, retention limits, redaction in logs.
- **Abuse:** rate limiting, bot detection, input validation, WAF rules, anomaly alerts, per-tenant quotas.
- **Multi-tenancy:** isolation boundaries (schema/DB/cluster), noisy neighbor controls.

11) Observability & operations (what strong candidates mention)

- **Golden signals:** latency, traffic, errors, saturation (per service + dependency).
- **Tracing:** correlation IDs across services; structured logs; sampled payload logging with PII guardrails.
- **SLIs/SLOs:** define + error budget; alert on symptoms not causes.
- **Runbooks:** dashboards, common failure modes, rollback steps, paging thresholds.

12) Deployment & data migrations (pros/cons)

- **Blue/green:** +fast rollback; –double capacity.
- **Canary:** +safe gradual rollout; –needs good metrics + routing controls.
- **Feature flags:** +decouple deploy from release; –flag debt.
- **DB migrations:** expand/contract; dual writes carefully; backfills throttled; verify before cutover.
- **Disaster recovery:** backups + restore drills; multi-region active-passive vs active-active tradeoffs.

How to “sprinkle” this in non-system-design rounds

- When you propose any pipeline, add: **API contract**, **data store**, **cache**, **async job**, **monitoring**, **rollback**.
- Use mini tradeoffs: “I’ll start with Postgres + Redis; if QPS grows, shard by user_id and move heavy queries to read model/search index.”
- End with ops: “Canary 1%, watch p99 + error rate + business metric; if regression → auto rollback; alert thresholds + dashboard.”