System Design Interview Framework: ‘Build X’

0. Real-World Framing (1 min)

- **Purpose**: Acknowledge the gap between interview exercises and real-world system design.
- **Key Point**: In practice, you design systems with the team you have. Familiarity, existing tooling, and organizational risk tolerance matter more than textbook architectures.
- **Suggested Opening Statement**

> “This interview is designed to evaluate my fluency in system design, familiarity with common technologies, and ability to weigh trade-offs. But in the real world, architecture choices are grounded in what your team knows and has the capacity to maintain. So while I’ll explore leading practices and trade-offs, I’ll stay grounded in pragmatic choices a team could realistically execute.”

---

1. Scope & Assumptions (5 min)

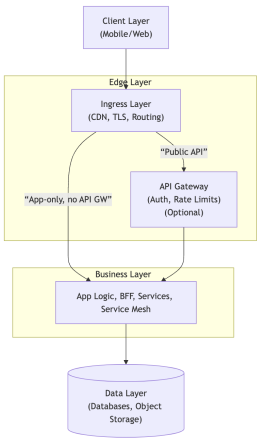
- 1.1 **Business Motivation**
  - What problem are we solving, and why now?
  - Strategic objectives and key stakeholders.
  - Organizational success criteria (e.g. revenue uplift, cost reduction, user growth).
- 1.2 **Users & Use Cases**
  - Primary personas and their goals.
  - Core journeys (MVP vs. full feature set).
  - Platform constraints (web, iOS, Android, public API).
- 1.3 **Budget & Monetization**
  - Phase-appropriate budget limits (MVP vs. scale).
  - Revenue models: ads, subscriptions, transactions.
  - Unit-economics impact of design choices (cost per user/transaction).
- 1.4 **Design & Optimization Goals**
  - Performance targets (P50/P95 latency, throughput).
  - Reliability commitments (SLAs, error budgets).
  - Scalability thresholds (peak users/day, requests/sec).
- 1.5 **Success & Failure Metrics**
  - Technical KPIs: latency percentiles, availability, error rate.
  - Business KPIs: conversion, engagement, retention.
  - Operational health: MTTR, incident count.
  - Cost efficiency: infrastructure cost per user/transaction.

- 1.6 **Other Considerations**
    - Internationalization & localization.
    - Data privacy & regulatory compliance (GDPR, HIPAA).
    - Supported versions/platforms and upgrade strategy.

---

2. High-Level Architecture (3 min)

> Before diving into specific components, it's helpful to align on the high-level architecture and how core traffic flows through the system.



- 2.1 **Tiered Architecture (Four-Tier Model)**
  Client → Edge → Business → Data
    - **Client Tier**
        - Mobile/Web clients handle UI rendering, local caching, and request shaping.
    - **Edge Tier**
        - Ingress: CDN, TLS termination, L7 routing, basic caching.
        - API Gateway (optional): Authentication, rate-limits, quotas, request shaping.
    - **Business Tier**
        - BFFs (Backends for Frontends): Client-specific adapters, payload aggregation.
        - Core Services: Domain logic, orchestration, long-running jobs.
        - Service Mesh: mTLS, retries, circuit-breaking, observability.
    - **Data Tier**
        - Databases, object storage, queues, and external APIs.
- 2.2 Request Flow
    1. **Client** issues a request →
    2. **Ingress** handles TLS, routing & CDN cache →

3. *(Public API only)* **API Gateway** enforces auth & quotas →
4. **BFF** transforms and aggregates →
5. **Core Services** run business logic (sync or via events) →
6. **Data Tier** persists/retrieves state →
7. Response returns via the same path.

---

3. Frontend (3 min)

- 3.1 **Platforms**
  - Web: React, Vue, Angular, Next.js.
  - Native: Swift (iOS), Kotlin/Java (Android).
  - Cross-platform: React Native, Flutter, .NET MAUI.
- 3.2 **Data Fetching**
  - REST vs. GraphQL based on payload flexibility.
  - Use REST for external/public APIs and gRPC for internal service-to-service calls that benefit from type safety and lower overhead.
  - Polling for semi-real-time updates.
  - WebSockets or SSE for persistent two-way communication.
- 3.3 **API Design Principles**
  - Use REST for straightforward resource-based APIs and widespread tooling compatibility.
  - Use GraphQL for complex data-fetching needs where clients benefit from flexible payload shaping and bundling.
  - Follow consistent versioning strategy (e.g., URI versioning or headers).
  - Define pagination, filtering, and sorting behaviors explicitly.
  - Treat APIs as products: document clearly, support deprecation, and consider developer experience (DX) across web, mobile, and partner clients.
- 3.4 **Client-Side Caching & Offline Support**
  - In-memory storage and local persistence (IndexedDB for web; AsyncStorage or SQLite for mobile/main apps).
  - Offline queue for writes and background sync.
  - Cache invalidation strategies (stale-while-revalidate, TTL).
- 3.5 **Session Management & Notifications**
  - Token storage & refresh patterns (cookies, localStorage).
  - Secure handling of refresh tokens (HttpOnly cookies, secure storage).
  - Push notifications setup (APNs, FCM) and in-app notification handlers.
- 3.6 **Ownership**
  - Typically owned by frontend or full-stack engineering squads.
  - Collaborates closely with UX, product, and backend/BFF teams.

---

4. Edge / Gateway Layer (3 min)

- 4.1 **Purpose and Role**: This layer sits between your clients and your backend systems, providing security, traffic management, and protocol translation so frontend and backend teams can iterate independently.

- 4.2 **Edge Components**
  - **Ingress Proxy**
    - CDN & Global Load Balancer (e.g. Fastly, CloudFront).
    - TLS termination, health checks, basic L7 routing.
  - **API Gateway** *(optional for public APIs)*
    - Authentication & authorization (JWT/OAuth introspection, API keys).
    - Rate limiting, IP allow/deny lists, request validation (schema, size).
    - Request shaping (body size quotas, header transformations) .
  - **Cross-Cutting Edge Services**
    - Feature-flag evaluation (LaunchDarkly, Flagsmith).
    - Token bootstrapping for service-mesh sidecars.
    - Edge compute functions (Lambda@Edge, Cloudflare Workers).

- 4.3 **Core Responsibilities**
  - **Security and Compliance**
    - Enforce authN/authZ before any backend logic runs.
    - Protect against DDoS, OWASP Top 10, and IP-based threats like IP spoofing.
  - **Traffic Management**
    - Rate limits, quotas, circuit-breaking at the edge.
    - Canary or blue/green routing for safe deployments.
      See Section 8.2 ("Deployment Strategies") for additional detail.
  - **Request Routing & Protocol Translation**
    - Route to the correct backend service or BFF.
    - Translate protocols (HTTP⇄gRPC, WebSocket upgrades).
  - **Payload Mapping & Shaping**
    - Fan-out or aggregate requests.
    - Rename fields, slim payloads, inject metadata.
  - **Performance Optimization**
    - Cache static and dynamic content at CDN edges.
    - Offload simple compute (geolocation, A/B logic) to edge workers.

- 4.4 **Ownership**: Platform team or shared between frontend/backend leads.

---

5. Backend Core Services (10 min)

- 5.1 **Frontend-Integration Layer (BFF)**
  - Acts as the “true backend edge” for each client type (web, mobile, partner).
  - Tailors payloads, aggregates across domain services, and translates protocols (e.g. gRPC → REST).

- Enforces any client-specific caching, request shaping, or retry logic before hitting core services.

- 5.2 **Business Domain Services**
  - Examples: Orders, Messages, Trips, Profiles.
  - Each service owns a cohesive set of entities and logic, exposing internal APIs for orchestration.

- 5.3 **Service Mesh (Sidecar-Based Networking)**
  - Provides mTLS, circuit-breaking, retries, and distributed tracing for all inter-service calls.
  - Enforces network-level policies (ACLs, rate-limits) consistently, independent of service code.
  - Offloads common concerns so each service—plus the BFF layer—can remain lean.

- 5.4 **Cross-Cutting Platform Services**
  - Shared infrastructure for auth, notifications, billing, metrics, experiments.
  - Some responsibilities (e.g. rate-limiting, token validation) may run at the edge or in BFF; heavier or async logic lives here.

- 5.5 **Architecture Styles**
  - Monolith, modular monolith, or microservices: each can host BFFs and sidecars.
  - With microservices, you typically deploy BFFs and sidecars in the same cluster, but route traffic via the ingress/API gateway.
    See generally Monolith to Microservices.

- 5.6 **Communication Patterns**
  - Synchronous RPC (REST, gRPC) for BFF→Domain and Domain→Domain calls, all secured by the mesh.
  - Async Messaging (Pub/Sub, Queues) for background workflows, decoupled from client-facing BFFs.

- 5.7 **Stateless vs Stateful Services**
  - BFFs and most domain services should be stateless to scale easily.
  - Stateful systems (sessions, game state) require sticky sessions or external state stores.

- 5.8 **Infrastructure Patterns**
  - Sidecar proxies (Envoy/Istio) alongside every service pod, including BFF deployments.
  - Mesh control plane on its own tier, speaking to sidecars across all services.

- 5.9 **Trade-Offs**
  - BFF adds per-client code but simplifies core services.
  - Service Mesh gives consistency and security but adds operational complexity.

- 5.10 **Communication Style: Request vs. Event Driven**
  - Request-Driven (e.g., REST, gRPC): Synchronous RPC-style communication between services. Well-suited for real-time flows where immediate feedback is needed, such as user actions or

transactional updates.
  - Event-Driven (e.g., queues, Pub/Sub, event streams): Asynchronous messaging used for background jobs, system decoupling, and resilience under load. Often backed by Apache Kafka, Apache Pulsar, or cloud-native equivalents.
  - Common Patterns:
    - Core services often use request-based RPC, while async workflows (e.g., billing, notifications, ML pipelines) are handled via events.
    - Events are also used to power downstream consumers: audit logs, analytics, or materialized views.
  - Most systems combine both: sync for control paths, async for data and side effects.

- 5.11 **Ownership**

  - BFFs often owned by the feature or frontend-platform team.
  - Core services by domain teams.
  - Mesh control plane by the platform/SRE team.

---

6. Data Storage & Retrieval (5 min)

- 6.1 Data Store Categories & Limitations

  - **Relational (SQL)**: PostgreSQL, MySQL
    - Use: ACID transactions, complex joins, ad-hoc queries.
    - Limitations: Challenging to scale horizontally (sharding/cluster required); heavy schema migrations.
  - **Key–Value Stores**: Redis, DynamoDB
    - Use: Ultra-low-latency get/put for sessions, caches, user state.
    - Limitations: No rich query or relationship support; minimal indexing.
  - **Document Stores**: MongoDB, Couchbase
    - Use: JSON-style documents, flexible/evolving schemas.
    - Limitations: Limited joins/transactions; risk of denormalization and data duplication.
  - **Column-Family Stores**: Cassandra, HBase
    - Use: High write throughput, wide-column models, time-series data.
    - Limitations: Eventual consistency by default; operational complexity (compaction, repair).
  - **Graph Databases**: Neo4j, Amazon Neptune
    - Use: Deep relationship traversals (social graphs, recommendations).
    - Limitations: Poor horizontal write scalability; specialized query language.
  - **Time-Series DBs**: InfluxDB, TimescaleDB
    - Use: Efficient ingestion & querying of timestamped metrics/events.
    - Limitations: Needs retention/downsampling; storage bloat risk.
  - **Search & Analytics Engines**: Elasticsearch, Meilisearch, Typesense
    - Use: Full-text search, faceted filtering, aggregations.
    - Limitations: Eventual consistency; heavy index rebuilds; high resource use.

- 6.2 When to Choose Which Store

  - **SQL**: Transactions + joins + strong consistency.

  - **Key–Value**: Simple lookups + caching + session data.
  - **Document**: Semi-structured or rapidly evolving schemas.
  - **Graph**: Complex many-to-many relationships, deep traversals.
  - **Time-Series**: High-frequency timestamped writes, windowed queries.
  - **Search**: Text search, analytics, log exploration.
    Caution: *Don't pick NoSQL merely to avoid schema design: poor data models create hidden debt.*

- 6.3 CAP Theorem & ACID Trade-Offs

  - **CAP Theorem** (under network partition)
    - CP: Consistency + Partition tolerance (sacrifice availability).
    - AP: Availability + Partition tolerance (sacrifice consistency).
    - CA: Consistency + Availability (no partition tolerance).
      See also this post for a diagrammatic discussion of the three CAP theorem options.
  - **ACID Transactions**
    - Atomicity: All or nothing; if any part of the transaction fails, roll back.
    - Consistency: Valid state transitions; all data integrity constraints and business rules are maintained throughout the transaction.
    - Isolation: No interference among concurrent transactions. Changes made by one transaction are not visible to other transactions until the first transaction is committed.
    - Durability: Persistence through failures; changes are permanently stored in the database, even if the system fails or crashes.

- 6.4 Blob Storage for Media

  - S3, GCS, Azure Blob: object stores for large binaries or static assets (fronted by CDN).

- 6.5 Search Layer

  - Elasticsearch, Meilisearch, Typesense: specialized engines for text search and analytics; evaluate SaaS vs. self-hosted.

- 6.6 Ownership

  - Shared among application teams (schema design, queries), data engineering (ETL/pipelines), and platform/infrastructure teams (operations, scaling).

---

7. Scalability, Reliability & Performance (5 min)

> When designing for scale, it's useful to distinguish between **vertical scaling** (making a single instance more powerful) and **horizontal scaling** (adding more instances). Most modern systems aim to scale horizontally for long-term flexibility, but both strategies have a role.

- 7.1 **Vertical Scaling (Scale-Up)**

  - Add more CPU, RAM, or IOPS to a single node.
  - Quickest path to performance gains in early-stage systems.
  - Simple to implement, but limited by hardware ceilings and diminishing returns.

- 7.2 **Horizontal Scaling (Scale-Out)**
  - Add more nodes behind a load balancer (e.g., web servers, DB replicas).
  - Often requires stateless services, sharded databases, or eventual consistency models.
  - Enables fault tolerance and high availability but increases architectural complexity.
- 7.3 **Reliability Techniques**
  - Autoscaling triggers & constraints (CPU, memory, custom metrics like queue length).
  - Scale-in safeguards (to avoid thrashing during transient load spikes).
  - Health checks (readiness and liveness probes) are critical for detecting broken or overloaded instances and safely removing them from load balancers.
  - Auto-Healing mechanisms (e.g., in Kubernetes or with auto-scaling groups) restart unhealthy services automatically without human intervention.
  - Ensure retries are safe and idempotent for critical operations (e.g., payments, orders); use idempotency keys or deduplication when needed.
  - Chaos engineering (e.g., Chaos Monkey-style fault injection) to proactively validate fail-over paths.
    See Principles of Chaos Engineering for an overview.
  - Multi-Region Deployment protects against entire region outages and reduces latency for global users, but adds complexity in data replication, failover routing, and consistency guarantees.
  - Defining SLIs/SLOs (latency P95, error budgets) as the foundation for any auto-scale or circuit-breaker policies.
- 7.4 **Performance Optimizations**
  - Caching: CDN, Redis, application-layer caching.
  - Backpressure and Resiliency: queuing, retry (with backoff), and circuit breakers
  - Query Tuning, Connection Pooling, Compression.
  - Load/stress testing (e.g. Gatling, JMeter) to validate horizontal-scale behavior before prod rollouts.
- 7.5. Ownership: Typically shared between SRE, infrastructure teams, and performance-focused engineers within product teams.

---

8. Deployment, Observability, Security & Ownership (as time allows)

- 8.1 **CI/CD & Infrastructure as Code**
  - Automate build, test, and deployment (e.g., GitHub Actions, CircleCI).
  - Manage infra as code (e.g., Terraform, Pulumi) and configuration as code (Ansible, Helm, Packer).
- 8.2 **Deployment Strategies**
  - Blue/Green: instant rollback, doubled infra cost.
  - Canary: phased rollout, relies on robust metrics and traffic control.
    Safe deployments rely on robust pre-prod validation and test environments — see Section 9.9

("Testing and Pre-Prod Environments") for guidance on staging, load testing, and end-to-end validation.

- 8.3 **Observability & Alerting**
    - Metrics, logs, and traces (e.g., Prometheus, Grafana, OpenTelemetry).
    - Dashboards, SLIs/SLOs, and error budgets.
    - Alerting and on-call workflows (e.g., PagerDuty, Opsgenie).
- 8.4 **Security Controls**
    - Authentication & authorization (OAuth 2.0, RBAC).
    - Secrets management (e.g., HashiCorp Vault, AWS Secrets Manager).
    - Network security (e.g., firewalls, mTLS).
    - Encryption in transit (TLS) and at rest (AES-256).
    - Supply chain integrity (image signing Notary v2, vulnerability scanning Trivy).
- 8.5 **Ownership**
    - CI/CD & IaC: DevOps/Platform teams.
    - Observability: SRE/Infrastructure teams.
    - Security: Security team with feature-team integration.

---

9. Extensions, Trade-Offs, Bottlenecks & Evolution (as time allows)

- 9.1 **Extensions & Integrations**
    - **AI/ML Services**
        - Traditional models: recommendations, fraud detection, anomaly detection.
        - Generative AI: LLM-based summarization, chatbots, content generation.
    - **Vector & Semantic Search**
        - Embedding pipelines, vector databases (Pinecone, Milvus) for similarity search.
    - **External APIs & SaaS Integrations**
        - Payment gateways, geolocation, identity providers, mapping, social logins.
    - **Event-Driven Pipelines**
        - Streaming analytics, real-time notifications, data pipelines (Apache Kafka, Apache Pulsar).
        - **Geographic Scaling**: multi-region failover, data residency, latency optimization.
        - **Privacy & Compliance**: PII separation, encryption requirements, GDPR/HIPAA constraints.
- 9.2 **Common Bottlenecks & Mitigations**
  See Sections 7 (“Scalability, Reliability & Performance”) and 6.1 (“Data Store Categories & Limitations”) for patterns on write limits, cache pressure, queue backlogs, and latency mitigation.
- 9.3 **Migration & Evolution Strategies**
    - **Monolith to Microservices**: incremental extraction (strangler-fig pattern).
    - **Modular Monolith**: adopt modules before splitting into services.
    - **Schema Versioning**: backward/forward-compatible migrations, dual writes.

- **Feature Toggles**: safe rollout, canary releases, controlled deprecation.

- 9.4 **Trade-Offs & Failure Modes**
  Covered in Sections 6.3 ("CAP Theorem & ACID Trade-Offs"), 7.4 ("Performance Optimizations"), and 8.4 ("Security Controls").

- 9.5 **System Evolution Patterns**
  - Feature toggles, schema versioning, modularization over time.

- 9.6 **Design Trade-Off: Event-Driven or Not?**
  - **Interviewer Prompt**: "Should your design use event-driven architecture? If not, why not?"
    - This is a common axis for evaluating trade-off awareness. The right answer depends on the use case, desired coupling, and latency/throughput profile.
  - **Choose Request-Driven When**:
    - Simplicity, traceability, and strong consistency are critical.
    - You need immediate feedback (e.g., checkout flows, profile edits).
  - **Choose Event-Driven When**:
    - Loose coupling, resiliency, or horizontal scalability is needed.
    - You're processing background work, derived data, or triggering workflows across domains.
  - **Communicate Your Rationale**: Explain where you've applied events and why. Highlight how you've handled failure (retries, backoff), observability, and delivery guarantees.

- 9.7 **Tenancy Models**
  - **Single-Tenant Systems**: Each customer has a dedicated instance of the application and its data. Offers strong isolation and simplified debugging but incurs higher operational cost.
  - **Multi-Tenant Systems**: Multiple customers share infrastructure, with logical data separation via tenant IDs. More cost-effective and scalable, but introduces complexity around data access, security boundaries, and performance isolation.
  - **Hybrid Models**: Mix of pooled and siloed tenants — e.g., shared app layer but isolated databases or compute for high-paying customers.
  - **Design Considerations**:
    - How is tenant identity propagated and enforced throughout the system? Apply fine-grained authorization (e.g., RBAC, per-object ACLs) where required.
    - What per-tenant quotas, rate limits, or billing hooks exist?
    - Can operational visibility be segmented by tenant?
  - Your architecture should account for tenant isolation, scaling patterns, and operational concerns from the beginning, even if you start single-tenant.

- 9.8 **Platformization & Developer Experience**
  - As systems and teams grow, consistency and velocity often become bottlenecks.
  - Platformization reduces friction by providing internal tools, templates, and abstractions that standardize core patterns (e.g., service creation, auth integration, metrics).
  - Developer experience (DX) initiatives improve build/run/debug loops and reduce context-switching overhead.
  - Common practices:

        - `create-service` CLIs or templates to scaffold new services with standard logging, tracing, metrics.
        - Shared SDKs for internal APIs (auth, config, billing).
        - Observability defaults: metrics, logs, alerts, and tracing pre-integrated into every service.
        - Self-service IaC: infra modules for queues, DBs, feature flags, storage.
    - A well-designed system should support internal reuse and allow teams to ship features safely and independently.

- 9.9 **Testing & Pre-Prod Environments**
    - Mature systems rely on multiple environments to validate changes safely before production.
    - Common environments include:
        - **Staging**: Mirrors production as closely as possible for full end-to-end validation.
        - **Integration/Test**: Used to validate contract compatibility and shared services across teams.
        - **Load Testing**: Simulates peak traffic and failure conditions to verify scaling limits.
        - **Synthetic or Ephemeral Environments**: Temporary, isolated deployments (e.g., per-PR or feature branch) for faster iteration.
    - **Testing Strategies**:
        - Unit tests for individual components.
        - Integration tests for service-to-service behavior.
        - End-to-end tests for full system workflows (often via UI automation or API smoke tests).
        - Contract tests (e.g., PACT) to detect breaking changes in upstream/downstream services.
    - Good pre-prod hygiene includes test data seeding, schema evolution checks, and observability parity with prod.