

IFRAILST System Design Framework – Full Candidate Template

Pillar	Focus	Key Outcome
I	Clarify scope & assumptions	Clear problem boundaries
F	Define functionality & constraints	Feature + SLA checklist
R	Quantify scale	Realistic design sizing
A	Draw core architecture	High-level diagram
I	APIs & data model	Versioned contracts
L	Handle latency & failure	Reliability strategy
S	Secure system	Auth & encryption plan
T	Make trade-offs & test	Measurable reasoning

I – Intent & Users

Why

To ensure you solve the *right problem*. Without understanding who uses the system, how often, and under what constraints, you risk building the wrong design.

What

This pillar defines **scope, scale, SLA, and assumptions**:

- Who are the users (internal/external)?
- What are their goals and access patterns?
- What scale and reliability are expected?
- Are there business or compliance constraints?

How

- Start your interview by asking 3–5 crisp clarifying questions.
- Use answers (or assumptions) to **set boundaries** for your design.
- State assumptions aloud if the interviewer says “assume”.
- Quantify scope (users, latency, region, retention).

 **Output:** Clear scope statement and quantified assumptions.

F – Functional & Non-Functional Requirements

Why

To confirm **what** the system must do (functional) and **how well** it must do it (non-functional). This ensures completeness and aligns design choices with real goals.

What

- **Functional:** Core system behaviors (CRUD, workflows, APIs, events).
- **Non-Functional:** Performance, reliability, security, compliance, latency, durability.

How

- Ask 3 questions on *functional* behavior (features, actions, edge cases).
- Ask 3 on *non-functional* goals (SLA, latency, consistency, compliance).
- Summarize before moving on.
- Identify the *non-negotiables* (e.g., “redirect must never fail”).

 **Output:** Prioritized list of must-have features and key performance constraints.

R – Rates, Scale & Growth

Why

All architectural decisions depend on **load, traffic pattern, and growth**. Misjudging scale leads to over- or under-engineered systems.

What

Defines quantitative expectations:

- Read/write QPS, concurrency.
- Data growth rate and retention.
- Cache hit ratios, traffic spikes, throughput.

How

- Estimate or derive order-of-magnitude metrics.
- Highlight read/write ratios and bottlenecks.
- Calculate monthly/yearly storage needs.
- Predict where scaling pressure first appears (cache, DB, I/O).

 **Output:** Quantified metrics driving data partitioning, caching, and scaling design.

A – Architecture & Boundaries

Why

© 2018–2025 FullStackMaster. All rights reserved.

YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | fullstackmaster.net

Architecture is the **core blueprint** showing how the system meets requirements. Boundaries define clear separation of concerns for scalability and resilience.

What

High-level components, their responsibilities, and communication paths:

- API layer, service layer, cache, DB, message queues, analytics.
- Component ownership and isolation.
- Failure boundaries and redundancy.

How

- Stop questioning, begin designing.
- State assumptions clearly (“I’ll assume global users and eventual consistency”).
- Draw an architecture diagram with 5–8 labeled boxes.
- Explain data flow from user to database.
- Mention technologies as *examples*, focus on principles (e.g., “distributed cache”, not “Redis” unless asked).

 **Output:** Clear, fault-tolerant component diagram with justification for each major choice.

I – Interfaces & Data Model

Why

© 2018–2025 FullStackMaster. All rights reserved.

YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | fullstackmaster.net

Defines how external systems, services, and users interact. Ensures extensibility and backward compatibility.

What

- APIs (REST, GraphQL, gRPC).
- Data schemas and keys.
- Contracts, idempotency, versioning, auth flow.

How

- Specify 3–4 core APIs (names, inputs, outputs).
- Design one data model table/entity with keys and attributes.
- Explain indexing, partitioning, and TTL logic.
- Include idempotency and schema evolution strategy.

 **Output:** Versioned API list + data model that supports all core operations.

L – Latency, Reliability & Resilience

Why

Every distributed system must handle **failures gracefully**. This pillar proves you can design for the *real world*.

What

Defines how the system meets latency goals and tolerates failures:

- p95/p99 latency targets.
- Failover, retries, caching.
- Circuit breakers, DLQ, backoff strategies.

How

- Quantify expected latency budgets (frontend + network + backend).
- Describe cache hit/miss flow.
- Outline retry policies and backpressure handling.
- Explain fallback modes (“serve from stale cache if DB fails”).
- Mention disaster recovery (RPO/RTO).

 **Output:** Performance numbers, retry logic, and recovery plan under failure.



S – Security

Why

© 2018–2025 FullStackMaster. All rights reserved.

YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | fullstackmaster.net

Without security, reliability is meaningless. This pillar ensures your design protects users and data at every layer.

What

- Authentication (AuthN) & Authorization (AuthZ).
- Data encryption (at rest, in transit).
- Rate limiting and abuse prevention.
- Auditing and PII protection.

How

- State how users authenticate (OAuth2, IAM).
- Define access control (role-based, attribute-based).
- Explain encryption strategy (TLS, KMS).
- Mention rate limits, logging, DDoS protection.

 **Output:** Secure data flow and clearly defined boundaries of trust.

T – Trade-offs, Testing & Telemetry

Why

© 2018–2025 FullStackMaster. All rights reserved.

YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | fullstackmaster.net

Every design involves trade-offs. This pillar demonstrates maturity, you can defend your choices and validate your system under stress.

What

- Comparisons (SQL vs NoSQL, sync vs async, monolith vs microservices).
- Monitoring and observability plan.
- Testing for scalability and resilience.

How

- Highlight 2–3 explicit trade-offs with rationale.
- Define key metrics to track (latency, error rate, cache hit ratio).
- Describe chaos/failure tests and load tests.
- Explain alerting and dashboard setup.

 **Output:** Balanced decision matrix + health monitoring and test plan.



How to Apply IFRAILST in an Interview

1. **Start with “I”,** clarify what’s being built.

2. **Move sequentially** through each pillar; 1–2 minutes each.
3. **Ask minimal but targeted questions** early.
4. **Assume missing info** confidently (“If not specified, I’ll assume X for this scale”).
5. **Draw architecture early**, refine it as you discuss.
6. **Always conclude with trade-offs and metrics**.