Rupesh Tiwari | Interview Coach | FullStackMaster
YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | +1 (609) 442-4081

🧭 IFRAILST System Design Framework – Full Candidate Template + Example

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.

Rupesh Tiwari | Interview Coach | FullStackMaster
YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | +1 (609) 442-4081

🟨 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

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

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

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

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.**

Rupesh Tiwari | Interview Coach | FullStackMaster
YouTube.com/@FullStackMaster | rupesh@fullstackmaster.net | +1 (609) 442-4081

🎯 System Design: URL Shortener (Bar-Raiser Conversational Example)

🟩 I – Intent & Users

Candidate:
"Before jumping in, I'd like to confirm the problem scope. Are we designing a public global shortener like Bitly or an internal company tool?"

Interviewer:
"Public — anyone can shorten URLs worldwide."

Candidate:
"Got it. Roughly what's the scale? How many users or requests per day?"

Interviewer:
"About 10K daily users, but during marketing events it can spike up to a million."

Candidate:
"Perfect. And the target SLA — something like 99.99% uptime and sub-100 ms redirects?"

Interviewer:
"Exactly."

Candidate:
"Okay, I'll assume global access, vanity URLs allowed, data retained 7 years, and reads dominate writes roughly 10 to 1. Sounds good?"

Interviewer:
"Yes."

🟨 F – Functional & Non-Functional

Candidate:
"Functionally, users need to:

1. Create a short URL for a given long URL.
2. Redirect short URLs to their long form.
3. Track click analytics like total hits and unique users.

Optionally, users might update or delete existing URLs."

Interviewer:
"Right."

Candidate:
"For non-functionals, I'll prioritize high availability (99.99%), low latency (<100 ms), strong durability for 7 years, and I'm okay with eventual consistency for writes since redirection is read-heavy."

Interviewer:
"Agreed."

🟧 R – Rates, Scale & Growth

Candidate:
"At peak 1 M users × 2 links/day = ~2 M new URLs/day → 25 writes/sec.
Reads are 10× heavier → ~250/sec average, 10 K QPS during campaigns.
Each record ~200 bytes → 2 TB/month, around 170 TB over 7 years.
So, the system's very read-heavy — caching will be critical to stay under 100 ms."

Interviewer:
"Makes sense."

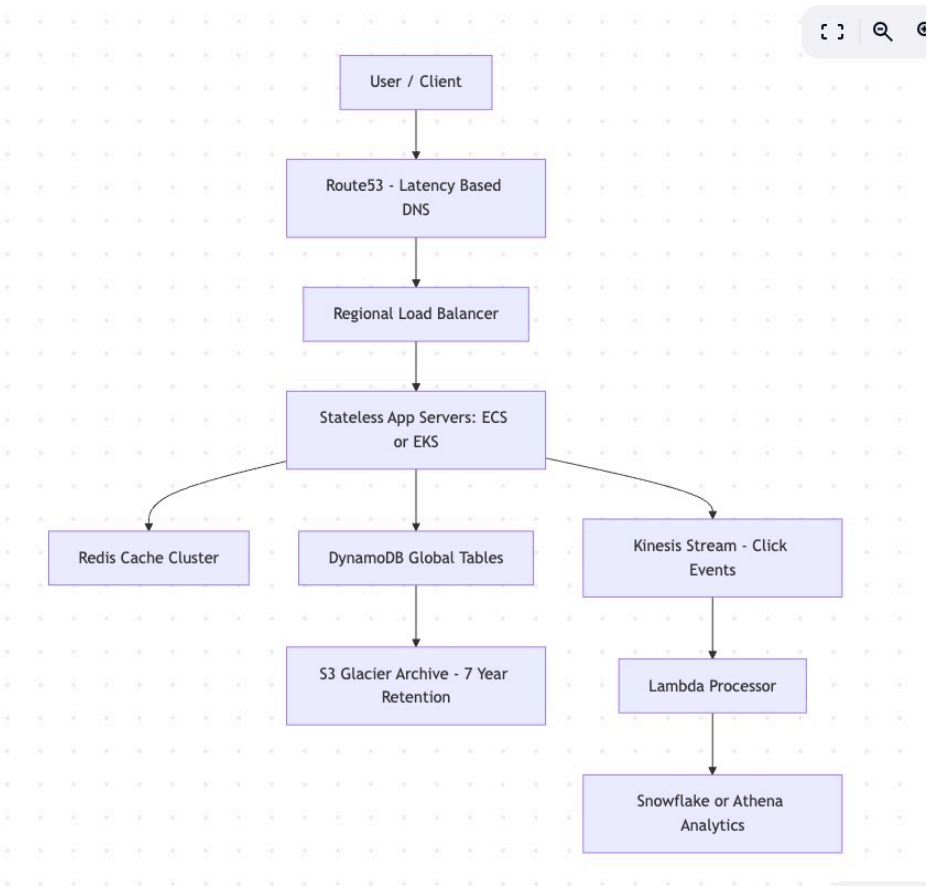
🟦 A – Architecture & Boundaries

Here's how I'd design a globally distributed, read-optimized system to meet the SLA.

Candidate:
"Let me walk through a global, low-latency setup."

```
graph TD
  A[User / Client] --> B[Route53 Latency-Based DNS]
  B --> C[Regional Load Balancer]
  C --> D[Stateless App Servers: ECS or EKS]
  D --> E[Redis Cache Cluster]
  D --> F[DynamoDB Global Tables]
  F --> G[S3 Glacier Archive - 7 Year Retention]
  D --> H[Kinesis Stream - Click Events]
  H --> I[Lambda Processor]
  I --> J[Snowflake or Athena Analytics]
```

View in diagram



Candidate:
"Users hit Route53, which routes to the nearest AWS region for latency.
Load balancer fans out to stateless app servers on ECS/EKS.
These servers handle short ID generation (Base-62, Snowflake pattern).
Hot short→long mappings live in Redis, persistent copies go into DynamoDB Global Tables, which replicate across regions.
Analytics events stream asynchronously via Kinesis → Lambda → Snowflake for dashboards.
Expired data moves automatically to S3 Glacier after 7 years."

Interviewer:
"Why Redis and not Memcache?"

Candidate:
"Memcache is cheaper but volatile. Losing cache means 3 s global latency spikes while repopulating. Redis gives persistence and replication — slightly costlier but SLA-safe."

🟪 I – Interfaces & Data Model

Candidate:
"The key APIs are straightforward:"

```
POST /v1/urls         {long_url, vanity?, expiry}
GET /v1/{short_code}  → 301 redirect
GET /v1/analytics/{id}
PATCH /v1/urls/{id}
DELETE /v1/urls/{id}
```

Candidate:
"For the data model:

- `short_code` is the primary key.
- Attributes: `long_url, user_id, created_at, expiry, click_count`.
- GSI on `user_id` for user dashboards.
- TTL handles expiry automatically."

Interviewer:
"How do you avoid collisions?"

Candidate:
"Each region gets its own ID namespace. I'd use Snowflake IDs encoded in Base-62. Collisions are astronomically rare — retry once if needed."

🟥 L – Latency, Reliability & Resilience

Candidate:
"To meet sub-100 ms redirect:

- Cache hit → 1 ms; miss → ~50–80 ms (DB fetch).
- Cache hit ratio target: 85 %.
- Multi-AZ Redis cluster, with replicas per region.
- Serve stale cache if DB is down.
- Retry writes with exponential backoff.
- Circuit breakers on DB calls.
- Pre-warm top 1 M links post-restart."

Interviewer:
"How fast do you recover after cache failure?"

Candidate:
"Within ~3 seconds Redis elects a replica. Cache pre-warm job runs asynchronously. User redirect latency may spike to 200 ms temporarily."

🛡️ S – Security

Candidate:
"OAuth2 for login; all APIs behind API Gateway with throttling.
TLS 1.3 for in-transit encryption, KMS for data at rest.
IAM roles scoped per service.
Rate-limit 100 req/s per IP or token.
Vanity URLs are salted to prevent enumeration."

Interviewer:
"Good — that's realistic and practical."

⚖️ T – Trade-offs, Testing & Telemetry

Candidate:
"Now, tying trade-offs to earlier requirements:"

- **Global latency <100 ms:** chose Route53 latency routing instead of round-robin; higher cost but consistent speed.
- **Scalability:** stateless ECS/EKS over monolith; +10 ms overhead but enables autoscale and 99.99% uptime.
- **Read-heavy load:** Redis instead of Memcache; costlier but persistent and multi-AZ safe.
- **Durability & locality:** DynamoDB Global Tables vs single region; accepts 300 ms async lag for 6 ms regional reads.
- **Redirect latency:** async analytics via Kinesis; 2–3 s lag in metrics but zero impact on redirect.
- **Retention cost:** move old data to S3 Glacier; saves 60% cost, slower retrieval okay for audits.

Testing:

- Load test 100 K QPS → p95 < 100 ms, <0.1% errors.
- Chaos test: kill Redis node → recover <3 s, regional failover <60 s.
- Soak test: 24 h steady load to check memory leaks.
- Alert if DB replication lag >500 ms.

Telemetry:

- Track latency, cache hit %, error rate, replication lag.
- Grafana dashboards per region.
- Alerts if p95 >150 ms for 5 min or cache hit <70%.
- Sample 0.1% of requests via X-Ray for tracing.

🧠 Candidate Wrap-Up

Candidate:
"This design guarantees <100 ms global redirects and 99.99% availability.
I traded 300 ms async replication lag for multi-region durability and user locality.
Redis caching ensures performance under load; DynamoDB and S3 cover durability.
Async analytics keep user experience fast.
We monitor every metric, inject chaos, and recover from failures within seconds.
It's simple, measurable, and cost-aware — ready for scale."

Interviewer:
"Excellent — that's a bar-raiser level answer. Every decision is tied to a requirement, metrics-backed, and pragmatic."