By

ABHISHEK BANSAL

SWE @ Google
Ex-Amazon, Microsoft,
D.E. Shaw

SYSTEM DESIGN CHEATSHEET – HLD GUIDE

A Practical Blueprint for Interviews and Real-World Design

Google amazon Microsoft D.E. SHAW

📘 Objective :

This document is a practical High-Level Design (HLD) Cheatsheet created after consistently solving and analyzing 50+ real-world design problems.

The goal is to help engineers:

✅ Understand key system components like databases, caching, load balancers, and queues

✅ Learn how to design scalable, fault-tolerant, and distributed systems

✅ Compare common architecture choices with trade-offs

✅ Use real interview-ready patterns, diagrams, and templates

✅ Build design intuition for cracking interviews at top tech companies

Whether you're preparing for FAANG-level interviews or want to level up your architecture skills — this guide is structured to be your go-to reference.

Linkedin : ABHISHEK BANSAL

System Design Foundations

	Concept	Explanation
🧠	High-Level Design (HLD)	Architectural blueprint of a system – defines major components, their interactions, scalability, reliability, and tech choices.
🎯	Why HLD is needed?	- Sets the technical vision - Allows team collaboration - Prepares for NFRs (Latency, Availability) - Helps in modular development
🚀	Key Goals	Scalability, Fault Tolerance, Cost Efficiency, Maintainability, Simplicity
📋	Key Deliverables	- Component diagrams - Tech stack overview - Data flow (Sync/Async) - API contracts (basic) - Database design (High level)

Load Handling & Scaling

Term	Description	Example
Load Balancer	Distributes traffic across servers	NGINX, AWS ELB
Horizontal Scaling	Add more machines to scale	Microservices on Kubernetes
Vertical Scaling	Increase power of one machine	Upgrade CPU/RAM
Stateless vs Stateful	Stateless: no user session Stateful: stores user context	Stateless APIs are easier to scale
Capacity Planning	Estimating future load using traffic, CPU, memory metrics	Required for autoscaling triggers

Data Flow & Communication Patterns

Pattern	Use Case	Tools
Synchronous	Immediate response required	REST, gRPC
Asynchronous	Delay acceptable, decoupled services	Kafka, SQS, RabbitMQ
Pub/Sub	One-to-many message broadcasting	Redis Streams, Google Pub/Sub
Event-Driven	Services react to events	Kafka + Event Handlers

Databases & Caching

Feature	Relational (SQL)	Non-Relational (NoSQL)
Structure	Tables with schema	JSON, key-value, wide-column
Best For	Joins, Transactions	High throughput, Flexibility
Examples	PostgreSQL, MySQL	MongoDB, DynamoDB, Cassandra

🔄 Caching Strategy Comparison

Cache Layer	Use Case	Non-Relational (NoSQL)
App-side Cache	Same request multiple times	In-memory (LRUMap)
Distributed Cache	Frequently accessed DB reads	Redis, Memcached
Write-through Cache	Write to cache + DB together	Ensures consistency
Write-back Cache	Write to cache first, DB later	Low latency, but risk of data loss

API Design & Gateway

Principle	Tools
REST vs gRPC	REST: easy & readable gRPC: fast & binary efficient
Idempotency	Multiple same requests → same result
Versioning	Use /v1/ in endpoint to avoid breaking changes
Rate Limiting	Prevents abuse of APIs – fixed or sliding window
API Gateway	Acts as the single entry point – handles auth, rate limit, routing
Tools	Kong, NGINX, AWS API Gateway, Apigee

System Reliability & Redundancy

Concept	Description	Benefit
Replication	Multiple copies of same data	High availability
Sharding	Split DB horizontally by key	Handles large-scale data
Failover	Automatic switch to standby	Zero downtime
Health Checks	Monitor app/service health	Auto-recovery & load shift
Circuit Breaker	Stops cascading failures	Resilience under pressure

🛠️ Examples

- Master-slave DB = Replication
- Hash-based DB partitioning = Sharding
- Netflix Hystrix = Circuit Breake

Monitoring, Logging & Alerting

Layer	Tool Examples	Why Important
Monitoring	Prometheus, Grafana, Datadog	Track performance, latency, uptime
Logging	ELK Stack (Elasticsearch, Logstash, Kibana)	Debug, audit, and root cause analysis
Tracing	Jaeger, OpenTelemetry	End-to-end request tracking
Alerting	PagerDuty, OpsGenie	Notify SREs/devs on failure

📌 **Pro Tip: Always attach a trace ID to requests for easy debugging across microservices.**

Designing for Scale

Strategy	Description	When to Use
CDN	Caches static content at edge	Media, JS/CSS, global traffic
Content Hashing	Unique filename for assets	Improves caching
Backpressure	Reject or delay requests under load	Protects downstream services
Retry with Exponential Backoff	Retry failed requests with increasing delay	Network-based failures
Throttling	Limit user or client request rate	Protect APIs and resources

🛠️🧠 **Use combination of retries + circuit breaker + queues to make services fault-tolerant.**

Real-World Architecture Examples (Mini)

System	Key Design Points
URL Shortener	- Hashing + DB for mapping - Redis cache for fast lookup - Read-heavy system
Chat App	- Websockets for real-time - Kafka for async messages - Horizontal scaling for message servers
Instagram Clone	- Media storage in S3/CDN - Timeline = Read Optimization - Sharded DB for posts & users

🔍 Interview Tip: Always begin with requirements, define APIs, then focus on components, bottlenecks, and trade-offs.