Full-Stack Engineering Fundamentals Master Map

A resume-aligned revision document for TypeScript, React, Next.js, Node.js, databases, Redis/queues, Docker/AWS, observability, security, and system design.

Prepared for Shubhojeet Bera - Full-stack / AI-native product engineering track

How to use this document

Use this as the master topic map before deep-diving with research agents, mentors, or interview prep. Each topic has a simple two-line explanation plus keywords you can expand into notes, mock questions, diagrams, and mini-projects.

Resume alignment

Full-stack engineer focused on AI-native product systems: TypeScript, React, Next.js, Node/Express, PostgreSQL, MongoDB, Redis/BullMQ, Docker/AWS, monitoring dashboards, deployment infra, queues, sandboxes, and production reliability.

Area from resume	What this PDF emphasizes
Frontend: React, Next.js, TanStack Query, Zustand	Rendering model, state boundaries, App Router, caching, data fetching, forms, a11y, performance, testing.
Backend: Node.js, Express.js, APIs	Event loop, middleware, validation, auth, API contracts, async work, graceful shutdown, reliability.
Data: PostgreSQL, MongoDB, Vector DBs	Schema design, indexes, query planning, transactions, isolation, migrations, document modeling, pgvector.
Infra: Redis, BullMQ, Docker, AWS	Caching, queues, workers, retries, idempotency, Docker images/containers/networks, S3/CDN, deployment patterns.
Experience: monitoring dashboards, deployment infra, memory leaks	Observability, metrics/logs/traces, alerting, incident response, performance debugging, release safety.

Table of contents

- 19. 30-day full-stack revision plan
- 20. Interview checkpoints
- 21. Research prompts for agents/partners
- 22. Source trail and deep-reading map

Positioning note

You do not need to pretend to be a 5-year engineer. You need to revise like one: know fundamentals, tradeoffs, failure modes, observability, and how to connect code decisions to user impact.

01. JavaScript and TypeScript Core

Revise the language layer deeply enough to explain runtime behavior, async bugs, type safety tradeoffs, and maintainable API contracts.

Topic	Simple two-line brief	Deep-dive keywords
Execution Context	Every JS function runs inside an execution context containing variables, scope, and this-binding. Understanding it explains hoisting, closures, and why some bugs appear only at runtime.	*call stack, lexical environment, hoisting, this*
Call Stack	The call stack tracks active function calls in last-in-first-out order. Stack overflows, deep recursion, and sync blocking become easier to reason about once this is clear.	*stack frames, recursion, RangeError*
Event Loop	The event loop lets JS coordinate async work without blocking the main thread. Learn macrotasks, microtasks, promises, timers, and how Node adds extra phases.	*microtask queue, timers, poll phase, nextTick*
Promises and Async/Await	Async/await is syntax over promises, not true blocking code. The important part is error propagation, parallel vs sequential execution, and cancellation strategy.	*Promise.all, allSettled, race, AbortController*
Closures	A closure lets a function remember variables from the scope where it was created. This is the basis of hooks, memoization, callbacks, and many subtle memory leaks.	*lexical scope, stale closure, private state*
Prototypes and Classes	JavaScript uses prototypal inheritance under the class syntax. You should know enough to explain prototype chains, instance methods, and object shape performance.	*prototype chain, class sugar, hidden classes*
Module Systems	CommonJS and ES Modules differ in loading, exports, static analysis, and bundling behavior. This matters in Node/Next apps, build tools, and package interop errors.	*CJS, ESM, tree shaking, dynamic import*
TypeScript Structural Typing	TypeScript checks compatibility by object shape, not declared class identity. This makes TS flexible but requires discipline around domain models and API contracts.	*duck typing, assignability, excess properties*
Generics	Generics let you preserve type information across reusable functions and components. They are essential for API clients, repository layers, form helpers, and reusable hooks.	*generic constraints, inference, type parameters*
Union and Intersection Types	Unions model alternatives; intersections combine capabilities. Used well, they make state machines, API responses, and error handling safer.	*discriminated unions, exhaustive switch, never*
Type Narrowing	Narrowing teaches TS what a value really is after runtime checks. It is the bridge between uncertain external input and safe internal code.	*typeof, in, instanceof, custom guards*
Utility Types	Utility types transform existing types instead of duplicating them. Use them for DTOs, partial updates, public/private fields, and API payload variants.	*Pick, Omit, Partial, Required, Record*
Runtime Validation	TypeScript disappears at runtime, so external data still needs validation. Use schema validation for request bodies, env vars, webhooks, and LLM/tool outputs.	*Zod, Valibot, io-ts, schema contracts*
Error Modeling	A mature codebase distinguishes expected domain errors from unexpected system errors. Model errors explicitly instead of throwing	*Result type, custom errors, error boundaries*

Topic	Simple two-line brief	Deep-dive keywords
	random strings everywhere.	
Package Management	Lockfiles, semver, peer dependencies, and workspace boundaries affect reproducible builds. Debugging dependency issues is a real full-stack skill.	*npm, pnpm, yarn, semver, peer deps*

02. Browser, Web Platform, and HTTP Fundamentals

Build confidence around what actually happens between browser, network, server, CDN, and API.

Topic	Simple two-line brief	Deep-dive keywords
DNS Resolution	DNS turns a domain into an IP address before any HTTP request can happen. Latency, caching, TTLs, and misconfigured records affect real production availability.	*A record, CNAME, TTL, propagation*
TCP and TLS	TCP provides reliable ordered delivery; TLS encrypts the connection and verifies identity. Handshakes add latency, so reuse and HTTP/2 matter.	*three-way handshake, certificates, ALPN*
HTTP Request Lifecycle	A browser request goes through DNS, TCP/TLS, request headers, server processing, response headers/body, parsing, and rendering. Know the complete flow end to end.	*request line, headers, body, response*
HTTP Methods	GET reads, POST creates/actions, PUT replaces, PATCH partially updates, DELETE removes. Correct method choice improves caching, idempotency, and API predictability.	*safe methods, idempotent methods, REST*
Status Codes	Status codes are API communication contracts, not decoration. 2xx succeeds, 3xx redirects, 4xx client issue, 5xx server issue.	*200, 201, 204, 400, 401, 403, 404, 409, 429, 500*
Headers	Headers carry metadata for auth, caching, content type, compression, tracing, and security. Many production bugs are header bugs.	*Authorization, Content-Type, ETag, Cache-Control*
Cookies	Cookies are browser-managed storage sent with matching requests. Secure, HttpOnly, SameSite, domain, and expiry flags control risk and behavior.	*session cookies, SameSite, CSRF*
CORS	CORS is browser-side permission for cross-origin reads; it is not server-to-server security. Preflight requests and credentialed requests are the common pain points.	*origin, preflight, Access-Control-Allow-Origin*
Browser Storage	localStorage, sessionStorage, IndexedDB, and cookies solve different persistence problems. Choose based on size, sync/async access, security, and browser behavior.	*IndexedDB, storage quota, XSS risk*
HTTP Caching	Caching uses headers like Cache-Control, ETag, Last-Modified, and CDN behavior to avoid repeated work. Wrong caching can create stale data or huge cost.	*max-age, stale-while-revalidate, ETag*
CDN Basics	A CDN caches content closer to users and reduces origin load. Understand cache keys, invalidation, signed URLs, and edge behavior.	*CloudFront, cache key, invalidation*
WebSockets	WebSockets keep a persistent bidirectional connection for low-latency updates. They are useful for chat, live dashboards, and agent/build progress streams.	*connection lifecycle, heartbeat, backpressure*
Server-Sent Events	SSE is a simpler server-to-browser streaming channel over HTTP. Great for progress updates, notifications, and AI token streaming when client-to-server messages are not needed.	*EventSource, retry, text/event-stream*
Webhooks	Webhooks are callbacks from external systems to your server. The hard parts are signature verification, retries, idempotency, ordering, and replay protection.	*HMAC, event IDs, retry, dead letter*

Topic	Simple two-line brief	Deep-dive keywords
Progressive Enhancement	A robust frontend still works reasonably when JS, network, or API calls fail. This mindset improves forms, SSR pages, and degraded states.	*HTML first, fallback, graceful degradation*

03. React Fundamentals and Frontend Architecture

Go beyond using React: understand rendering, state boundaries, memoization, effects, performance, and maintainable component architecture.

Topic	Simple two-line brief	Deep-dive keywords
Component Model	React apps are built by composing reusable components. The real skill is choosing component boundaries that match product behavior and reduce hidden coupling.	*composition, props, children, slots*
Rendering Model	React calls components to calculate UI, then reconciles changes into the DOM. Rendering is not the same as committing DOM updates.	*render phase, commit phase, reconciliation*
Virtual DOM / Reconciliation	React compares previous and next element trees to update only what changed. Keys are crucial because they tell React how list items map across renders.	*diffing, keys, identity*
State	State is local memory that triggers re-render when changed. Put state as close as possible to where it is used, then lift only when sharing is required.	*useState, derived state, lifting state*
Props	Props are read-only inputs from parent to child. Treating props as contracts keeps components predictable and testable.	*prop drilling, component API*
Effects	Effects synchronize React with external systems like network, DOM APIs, subscriptions, and timers. Most effect bugs come from wrong dependencies or doing derivation inside effects.	*useEffect, cleanup, dependency array*
Stale Closures	A stale closure happens when a callback reads old state from an older render. You see it in intervals, subscriptions, async handlers, and debounced callbacks.	*refs, functional updates, dependency correctness*
Refs	Refs hold mutable values without causing re-render. Use them for DOM access, imperatively managed APIs, and latest-value escape hatches.	*useRef, forwarded refs, imperative handle*
Memoization	Memoization avoids repeated expensive work or unstable references. Use it deliberately; unnecessary memoization can make code harder without real performance benefit.	*useMemo, useCallback, React.memo*
Context	Context avoids prop drilling for cross-cutting data. It is not a universal state manager because broad context updates can re-render many consumers.	*provider boundaries, context splitting*
Client State vs Server State	Client state belongs to the browser UI; server state belongs to a backend and needs fetching, caching, invalidation, and synchronization. This distinction is why TanStack Query exists.	*Zustand vs TanStack Query, staleTime*
Zustand	Zustand is a small hook-based client state store. Use it for UI/session-like app state, not as a replacement for server data caching.	*stores, selectors, middleware*
TanStack Query	TanStack Query manages fetching, caching, synchronizing, and updating server state. Learn query keys, stale time, invalidation, optimistic updates, and hydration.	*queryKey, mutation, cache, invalidateQueries*
Forms	Forms need validation, submission state, errors, accessibility, and partial failure handling. Good forms are a product-quality signal, not a simple input exercise.	*controlled/uncontrolled, React Hook Form, zod*
Accessibility	Accessibility means semantic HTML, keyboard	*ARIA, tab order, labels, focus trap*

Topic	Simple two-line brief	Deep-dive keywords
	support, labels, contrast, focus states, and screen reader-friendly UI. It also improves code quality and testing.	
Frontend Performance	Measure bundle size, render cost, network waterfalls, image loading, and hydration. Lighthouse is a starting point, not the whole truth.	*Core Web Vitals, code splitting, lazy loading*
Memory Leaks	Leaks come from uncleaned subscriptions, timers, event listeners, observers, and retained references. Your Merlin work maps directly here.	*cleanup, heap snapshots, abort fetch*
Component Abstraction	Good abstractions reduce repeated behavior without hiding important state. Extract when duplication reveals a stable concept, not just because code looks long.	*headless components, compound components*
Frontend Testing	Test behavior, not implementation details. Component tests catch UI logic; E2E tests catch real user flows and integration problems.	*React Testing Library, Playwright, Cypress*

04. Next.js and Modern React App Architecture

Revise Next.js from routing, rendering, caching, server/client boundaries, auth, APIs, and deployment behavior.

Topic	Simple two-line brief	Deep-dive keywords
App Router Mental Model	The App Router organizes UI by nested layouts, pages, loading states, and error boundaries. Learn how routing maps to filesystem and how layouts preserve state.	*app directory, layout.tsx, page.tsx*
Server Components	Server Components run on the server and can fetch data without shipping that logic to the client. They reduce bundle size but cannot use browser-only APIs or client hooks.	*RSC, server/client boundary*
Client Components	Client Components run in the browser and support interactivity, state, effects, and event handlers. Use client boundaries intentionally because they increase client JS.	*use client, hydration, browser APIs*
SSR	Server-side rendering generates HTML per request. Useful for personalized/dynamic pages but requires attention to latency and caching.	*request-time rendering, dynamic rendering*
SSG	Static generation builds pages ahead of time. Great for stable content, marketing pages, docs, and pages that can be cached aggressively.	*build-time rendering, static output*
ISR	Incremental Static Regeneration updates static pages after deployment. It balances cache speed with eventual freshness.	*revalidate, stale content, regeneration*
Streaming	Streaming sends UI progressively instead of waiting for all data. This improves perceived speed for slow data or AI-like progressive responses.	*Suspense, loading.tsx, progressive rendering*
Next.js Caching	Next has multiple caching layers: fetch cache, route cache, router cache, and revalidation behavior. Many Next bugs are caching mental-model bugs.	*cache, no-store, revalidateTag, revalidatePath*
Route Handlers	Route handlers implement backend endpoints inside Next. Use them for BFF-style APIs, auth callbacks, webhooks, and lightweight server operations.	*GET/POST handlers, NextResponse*
Server Actions	Server Actions let forms/components call server functions directly. They simplify some mutations but require clear validation, auth, and error boundaries.	*use server, form actions, mutations*
Middleware	Middleware runs before route handling and is useful for auth redirects, rewrites, localization, and lightweight request checks. Keep it small because it can run frequently.	*edge runtime, matcher, redirects*
Image Optimization	Next Image improves image loading through resizing, lazy loading, and formats. Know when remote patterns, CDN, and cache behavior matter.	*next/image, remotePatterns, priority*
Environment Variables	Server env vars must not leak to client bundles; NEXT_PUBLIC variables intentionally do. Mismanaged envs cause serious security and deployment issues.	*runtime env, build env, secrets*
Authentication in Next	Auth in Next needs cookies/sessions, server checks, middleware routing, CSRF awareness, and API protection. Avoid trusting only client-side state.	*NextAuth/Auth.js, JWT, session cookies*
Deployment Runtime	Understand serverless, edge, and long-running	*Vercel limits, cold start, workers*

Topic	Simple two-line brief	Deep-dive keywords
	server constraints. Queues, webhooks, and separate workers are often needed for slow or stateful jobs.	

05. Backend Engineering with Node.js and Express

Revise backend fundamentals behind APIs, async work, security, validation, database access, and production request handling.

Topic	Simple two-line brief	Deep-dive keywords
Node.js Runtime	Node runs JS on V8 and uses libuv for async I/O. It is strong for I/O-heavy systems, but CPU-heavy work can block the event loop.	*V8, libuv, non-blocking I/O*
Express Middleware	Middleware is a chain of functions around a request and response. Auth, logging, parsing, validation, CORS, and error handling all fit this model.	*req, res, next, middleware order*
Routing	Routes map HTTP methods and paths to handlers. Good routing separates transport concerns from business logic and data access.	*router, controllers, route params*
Request Validation	Never trust request bodies, query params, headers, or external webhook payloads. Validate at the boundary and convert unknown data into typed domain input.	*schema validation, DTOs, sanitization*
Error Handling	Centralized error handling keeps APIs consistent and prevents leaking internals. Separate operational errors from programmer bugs.	*error middleware, status codes, logging*
Authentication	Authentication proves who the user is. It can use sessions, JWTs, OAuth, magic links, or SSO depending on product needs.	*sessions, JWT, OAuth, cookies*
Authorization	Authorization decides what an authenticated user can access. Model permissions near resources, not only near routes.	*RBAC, ABAC, ownership checks*
Pagination	Pagination avoids returning huge result sets. Offset is simple; cursor pagination scales better for changing datasets.	*offset, cursor, limit, created_at*
Idempotency	Idempotency ensures retrying the same operation does not duplicate side effects. This is critical for payments, webhooks, job submission, and agent actions.	*idempotency key, deduplication, safe retries*
Background Work	Slow work should leave the request path and run in workers. APIs enqueue jobs, workers process, and clients poll or receive events.	*BullMQ, Redis, job status, workers*
File Uploads	Uploads should handle size limits, content validation, storage location, signed URLs, malware risk, and cleanup. Direct-to-S3 is often better than routing large files through app servers.	*multipart, presigned URLs, S3, MIME*
API Versioning	APIs change over time, so versioning and backward compatibility matter. Avoid breaking clients casually; add fields before removing fields.	*v1/v2, additive changes, deprecation*
Configuration Management	Config should come from environment or secret stores, not hardcoded values. Validate config at boot so failures happen early.	*12-factor, env schema, secret rotation*
Dependency Injection	Dependency injection makes code testable by passing services explicitly. It prevents business logic from being tightly coupled to concrete DB or API clients.	*service layer, repositories, ports/adapters*
Graceful Shutdown	A production server should stop accepting new requests, finish in-flight work, close DB connections, and exit cleanly. This matters during deploys and crashes.	*SIGTERM, connection draining, health checks*

06. API Design: REST, GraphQL, WebSockets, and Contracts

Know how to design APIs that are predictable, evolvable, testable, and safe under retries and partial failure.

Topic	Simple two-line brief	Deep-dive keywords
REST Resource Modeling	REST APIs model resources with nouns and standard methods. Good resource design makes clients simple and status codes meaningful.	*resources, verbs, representations*
DTOs	DTOs define what crosses API boundaries. Keep DB models, domain models, and public API payloads separate when systems grow.	*request DTO, response DTO, mapping*
OpenAPI	OpenAPI documents HTTP APIs in a machine-readable format. It improves client generation, contract tests, and team communication.	*Swagger, schemas, examples*
GraphQL	GraphQL lets clients ask for exactly the data they need. It needs careful handling of N+1 queries, auth, depth limits, and caching.	*schema, resolver, DataLoader, fragments*
RPC	RPC-style APIs model actions instead of resources. They are practical for internal services, commands, workflows, and tool calls.	*tRPC, gRPC, command APIs*
BFF Pattern	Backend-for-Frontend adapts backend services to one frontend experience. It reduces frontend complexity and hides internal service shape.	*Next route handlers, aggregation layer*
Pagination Design	Cursor pagination is safer for feeds and changing lists. Offset pagination is fine for small admin tables and predictable datasets.	*cursor, nextToken, stable sort*
Filtering and Sorting	Filtering/sorting should be explicit, indexed, and validated. Unbounded dynamic filters can create slow queries and security issues.	*query params, indexes, allowlists*
API Error Shape	A consistent error body helps clients handle failures gracefully. Include code, message, fields, request ID, and safe details.	*problem+json, error codes, request ID*
WebSocket Protocol Design	WebSocket messages need event names, payload schemas, ack/retry behavior, auth, reconnect strategy, and versioning. Treat them like APIs, not random JSON.	*events, heartbeat, ack, reconnect*
SSE Protocol Design	SSE is simple but still needs event IDs, retry behavior, keep-alive comments, and resumability if reliability matters.	*Last-Event-ID, retry, keep-alive*
Webhook Contracts	A webhook endpoint must verify signatures, dedupe event IDs, handle retries, and return quickly. Process slow work asynchronously.	*signature, timestamp, replay window*
Backward Compatibility	Public APIs should evolve without breaking clients. Prefer additive fields, optional parameters, and explicit deprecation windows.	*semver, migration guide, versioning*

07. PostgreSQL, SQL, and Relational Database Depth

This is one of the biggest seniority multipliers: schema design, indexes, transactions, query plans, locks, migrations, and operational safety.

Topic	Simple two-line brief	Deep-dive keywords
Relational Modeling	Relational design models entities and relationships with tables, keys, and constraints. Good schema design prevents bugs before app code runs.	*tables, rows, PK, FK, constraints*
Normalization	Normalization reduces duplication and update anomalies. Denormalize only when reads justify it and consistency strategy is clear.	*1NF, 2NF, 3NF, denormalization*
Primary and Foreign Keys	Primary keys identify rows; foreign keys preserve referential integrity. They make data harder to corrupt and easier to reason about.	*cascade, restrict, references*
Indexes	Indexes speed reads by maintaining extra lookup structures. They also slow writes and consume storage, so index based on real query patterns.	*B-tree, GIN, GiST, partial index*
Composite Indexes	Composite indexes depend on column order and query predicates. Learn left-prefix behavior and how sort order interacts with indexes.	*multi-column index, leftmost prefix*
Query Planner	The planner chooses how to execute SQL based on stats and costs. EXPLAIN helps you see whether queries scan, index, join, or sort inefficiently.	*EXPLAIN ANALYZE, seq scan, index scan*
Transactions	A transaction groups operations into an all-or-nothing unit. It protects consistency when multiple writes must succeed or fail together.	*BEGIN, COMMIT, ROLLBACK, atomicity*
ACID	ACID means atomicity, consistency, isolation, and durability. It is the core language for explaining database correctness.	*atomicity, consistency, isolation, durability*
Isolation Levels	Isolation controls what concurrent transactions can observe. Read committed, repeatable read, and serializable trade correctness and concurrency.	*dirty read, phantom read, serialization failure*
Locks	Locks coordinate concurrent access. Know row locks, table locks, deadlocks, lock waits, and how long transactions hurt systems.	*SELECT FOR UPDATE, deadlock, lock timeout*
Connection Pooling	Databases cannot handle unlimited connections. Pooling reuses connections and protects DB capacity, especially in serverless environments.	*pgBouncer, pool size, max connections*
Migrations	Migrations evolve schema safely over time. Production migrations need backward compatibility, rollbacks, batching, and low-lock strategies.	*expand-contract, backfill, zero-downtime*
JSONB	JSONB gives flexible document-like fields inside Postgres. Use it for semi-structured data, but avoid replacing relational modeling everywhere.	*GIN index, jsonb_path_query*
Full Text Search	Postgres can do lexical search using tsvector and indexes. It is useful for keyword search and hybrid retrieval baselines.	*tsvector, tsquery, ranking*
pgvector	pgvector stores embedding vectors inside Postgres for semantic search. It is useful when you want app data and vector search in one operational system.	*HNSW, IVFFlat, cosine distance*

Topic	Simple two-line brief	Deep-dive keywords
Read Replicas	Replicas scale reads and improve resilience but introduce replication lag. Never assume a just-written row is instantly visible on a replica.	*primary/replica, lag, failover*
Backup and Restore	Backups are only real if restores are tested. Know snapshots, PITR, retention, and recovery objectives.	*RPO, RTO, WAL, PITR*

08. MongoDB and Document Modeling

Revise NoSQL tradeoffs so you can explain when MongoDB is useful and when relational modeling is safer.

Topic	Simple two-line brief	Deep-dive keywords
Document Model	MongoDB stores JSON-like documents instead of rows. It fits data that is naturally nested and usually read together.	*document, collection, BSON*
Embedding vs Referencing	Embed data when it is owned and read together; reference data when it is shared or grows independently. This is the key Mongo modeling decision.	*one-to-few, one-to-many, references*
Indexes in MongoDB	Mongo indexes support query speed but must match access patterns. Missing indexes often show up as slow collection scans.	*compound index, multikey, text index*
Aggregation Pipeline	Aggregation transforms and combines documents through stages. It is powerful but can become expensive if filtering/index usage is poor.	*$match, $group, $lookup, $project*
Schema Flexibility	Mongo allows flexible schema, but production apps still need validation and conventions. Schema-less does not mean design-less.	*Mongoose schemas, JSON schema validation*
Transactions in MongoDB	Mongo supports multi-document transactions, but using them everywhere may indicate poor modeling. Prefer single-document atomicity when possible.	*session, transaction, atomic updates*
ObjectId	ObjectId includes timestamp and uniqueness properties. Understand how it differs from UUIDs and why sorting by ObjectId can approximate creation order.	*ObjectId, UUID, createdAt*
Pagination in MongoDB	Skip/limit is simple but can be slow for deep pages. Cursor-style pagination using indexed fields scales better.	*range query, _id cursor*

09. Redis, Caching, Queues, and Realtime Systems

This section maps directly to your BullMQ, Redis-buffered writes, semantic caching, monitoring, and production reliability work.

Topic	Simple two-line brief	Deep-dive keywords
Redis Mental Model	Redis is an in-memory data store commonly used for caching, queues, locks, counters, and pub/sub. It is fast because data structures live in memory.	*strings, hashes, lists, sets, sorted sets*
Cache-Aside	The app reads cache first, falls back to DB, then writes result into cache. It is simple and common, but stale data and stampedes need handling.	*miss, fill, TTL, invalidation*
Write-Through Cache	Writes go through cache and then to database. It keeps cache warm but adds write-path complexity and failure handling.	*write path, consistency, latency*
Write-Behind Cache	Writes hit cache first and persist later. It improves latency but risks data loss unless durability and replay are designed carefully.	*buffered writes, flush, replay*
TTL Strategy	TTL prevents stale data from living forever. Choose TTL based on data freshness, cost, and invalidation complexity.	*expiry, jitter, stale-while-revalidate*
Cache Invalidation	Invalidation is hard because cached copies must be updated or removed when source data changes. Prefer explicit keys and predictable invalidation paths.	*delete-on-write, tags, versioned keys*
Cache Stampede	A stampede happens when many requests rebuild the same expired cache key at once. Fix with locks, request coalescing, jitter, or stale serving.	*dogpile, mutex, singleflight*
Distributed Locks	Locks coordinate work across processes but can be dangerous if expiry, ownership, and failure cases are ignored. Use them sparingly and make jobs idempotent anyway.	*SET NX PX, Redlock, fencing token*
BullMQ Basics	BullMQ stores jobs in Redis and lets workers process them asynchronously. Understand queues, workers, concurrency, retries, delays, priorities, and job states.	*Queue, Worker, QueueEvents, attempts*
Retry and Backoff	Retries recover from transient failures but can amplify outages if uncontrolled. Use exponential backoff, max attempts, jitter, and dead-letter handling.	*exponential backoff, jitter, DLQ*
Idempotent Jobs	A job may run more than once after crashes or retries. Design job handlers so repeated execution does not corrupt state or duplicate side effects.	*dedupe keys, checkpoints, upserts*
Dead Letter Queue	A DLQ stores jobs that failed permanently for later inspection or manual recovery. It is essential for production async systems.	*failed jobs, replay, alerting*
Pub/Sub	Pub/sub broadcasts messages to subscribers but does not usually guarantee durable delivery. Good for live updates, not critical workflows without persistence.	*channels, subscribers, lost messages*
Streams	Redis Streams provide append-only event logs with consumer groups. They are useful when you need more durability than simple pub/sub.	*XADD, consumer group, ack*
Realtime Notifications	Realtime systems need connection tracking, auth, heartbeats, fanout, offline state, and replay. WebSockets/SSE are only the transport piece.	*presence, fanout, delivery semantics*

10. Authentication, Authorization, and Security Fundamentals

You need enough security depth to design safe apps, protect APIs, and explain threat models in interviews.

Topic	Simple two-line brief	Deep-dive keywords
Password Hashing	Passwords must be hashed with slow adaptive algorithms, never encrypted or stored plainly. Salts prevent precomputed attacks.	*bcrypt, argon2, salt, work factor*
Sessions	Session auth stores server-side session state and sends a cookie to the browser. It is easy to revoke and strong for traditional web apps.	*session ID, Redis session store, cookie*
JWTs	JWTs are signed tokens containing claims. They are useful for stateless auth but hard to revoke and must be short-lived if risk is high.	*claims, exp, aud, iss, JWK*
Access vs Refresh Tokens	Access tokens are short-lived credentials for APIs; refresh tokens obtain new access tokens. Refresh token rotation reduces long-term theft risk.	*rotation, revocation, token theft*
OAuth 2.0	OAuth delegates authorization between apps without sharing passwords. Know authorization code flow, PKCE, scopes, and redirect URI validation.	*PKCE, scopes, redirect_uri*
RBAC	Role-based access control maps users to roles and roles to permissions. It is simple and good for admin/business apps.	*roles, permissions, hierarchy*
ABAC	Attribute-based access control uses user/resource/context attributes. It fits complex rules like team ownership, plan limits, or region constraints.	*policy, subject, resource, context*
XSS	Cross-site scripting lets attackers run JS in a user browser. Prevent with escaping, sanitization, CSP, and avoiding unsafe HTML injection.	*stored XSS, reflected XSS, CSP*
CSRF	CSRF tricks a browser into sending authenticated requests. SameSite cookies, CSRF tokens, and checking origins reduce the risk.	*SameSite, CSRF token, Origin header*
SQL Injection	SQL injection happens when user input becomes executable SQL. Use parameterized queries and never concatenate untrusted input.	*prepared statements, query parameters*
SSRF	Server-side request forgery tricks your server into requesting internal resources. Validate URLs, block private IPs, and proxy outbound fetches safely.	*metadata service, allowlist, DNS rebinding*
Rate Limiting	Rate limiting protects systems from abuse and overload. Choose keys, algorithms, storage, and failure behavior based on product risk.	*IP limit, user limit, token bucket*
Input Sanitization	Validation checks shape; sanitization cleans unsafe content. Use both depending on whether you reject or transform input.	*HTML sanitization, allowlists*
Secrets Management	Secrets should live in secret stores or environment config, not code. Rotate secrets, scope permissions, and audit access.	*AWS Secrets Manager, env vars, rotation*
Least Privilege	Every service/user should have only the permissions it needs. This reduces blast radius when a credential leaks or a service is compromised.	*IAM policy, scoped token, deny by default*
Security Headers	Headers can reduce browser attack surface. Learn CSP, HSTS, X-Frame-Options, Referrer-	*CSP, HSTS, frame ancestors*

Topic	Simple two-line brief	Deep-dive keywords
	Policy, and cookie flags.	

11. Docker, Deployment, and Cloud Fundamentals

Connect your Docker sandbox/deployment experience to general production deployment mental models.

Topic	Simple two-line brief	Deep-dive keywords
Docker Image	An image is a packaged filesystem and metadata used to start containers. Smaller, deterministic images improve security, speed, and reproducibility.	*Dockerfile, layers, base image*
Docker Container	A container is a running isolated process created from an image. It shares the host kernel but has isolated filesystem, network, and process view.	*namespaces, cgroups, lifecycle*
Docker Layers	Each Dockerfile instruction creates a layer that can be cached. Layer ordering affects build speed and image invalidation.	*COPY order, cache, multi-stage*
Multi-Stage Builds	Multi-stage builds separate build dependencies from runtime image. This reduces image size and attack surface.	*builder stage, runtime stage*
Volumes	Volumes persist data outside container lifecycle. Use them for database data and avoid storing important state only inside containers.	*bind mount, named volume, persistence*
Container Networking	Containers communicate over Docker networks and exposed ports. Understand localhost inside a container vs host localhost.	*bridge network, ports, DNS*
Docker Compose	Compose defines multi-container local environments. It is great for app + Postgres + Redis + worker development setups.	*services, depends_on, networks*
Environment Separation	Development, staging, and production should have separate config, data, credentials, and deployment flows. Bugs often come from environment drift.	*staging, prod parity, config*
CI/CD Pipeline	CI validates changes; CD ships them safely. Good pipelines run tests, build artifacts, scan, deploy, and rollback.	*GitHub Actions, artifacts, rollback*
Blue-Green Deployment	Blue-green keeps two production environments and switches traffic after validation. It reduces downtime and rollback risk.	*traffic switch, rollback, warmup*
Canary Deployment	Canary sends a small percentage of traffic to a new version first. It reduces blast radius and catches issues with real users.	*progressive rollout, metrics gate*
Serverless	Serverless runs code on demand with managed scaling but has cold starts, timeouts, and runtime constraints. Long jobs often need queues/workers.	*Lambda, Vercel functions, cold start*
Object Storage	Object storage stores files/blobs separately from app servers. Use it for user uploads, generated artifacts, logs, and static assets.	*S3, buckets, object keys*
Presigned URLs	Presigned URLs let clients upload/download directly to storage with limited permission and expiry. They reduce server load and improve scalability.	*signed upload, expiry, content type*
CDN with Object Storage	A CDN in front of storage improves global latency and reduces origin cost. Understand cache invalidation, signed URLs, and public/private access.	*CloudFront, cache policy, signed URL*
Infrastructure as Code	IaC defines cloud resources in code so infra is reproducible and reviewable. It reduces manual console drift.	*Terraform, CDK, Pulumi*

12. Observability, Reliability, and Production Debugging

This is where you can sound much more experienced than your YOE: logs, metrics, traces, SLOs, incidents, and debugging loops.

Topic	Simple two-line brief	Deep-dive keywords
Logging	Logs explain discrete events and failures. Good logs include request ID, user/service context, error details, and safe metadata.	*structured logs, log levels, correlation ID*
Metrics	Metrics track numeric system behavior over time. Use them for latency, errors, throughput, saturation, queues, cache hits, and business events.	*counter, gauge, histogram, p95*
Tracing	Tracing follows one request across services, DB calls, queues, and external APIs. It helps debug latency and distributed failures.	*spans, trace ID, OpenTelemetry*
Health Checks	Health checks tell orchestrators whether a service can receive traffic. Separate liveness from readiness to avoid bad restarts.	*liveness, readiness, startup check*
SLOs and SLIs	SLIs measure user-facing reliability; SLOs define targets. This shifts engineering from vibes to measurable reliability.	*availability, latency, error budget*
Alerting	Good alerts are actionable and tied to user impact. Bad alerts create noise and get ignored.	*paging, severity, alert fatigue*
Incident Response	Incidents need detection, ownership, mitigation, communication, and postmortem. Senior engineers reduce recurrence, not just fix once.	*runbook, postmortem, RCA*
Dashboard Design	Dashboards should answer: is the system healthy, where is it failing, and what changed. Your Iterate monitoring dashboard maps directly to this.	*golden signals, RED, USE*
Feature Flags	Feature flags decouple deployment from release. They enable gradual rollout, quick rollback, experiments, and kill switches.	*flags, targeting, kill switch*
Circuit Breakers	Circuit breakers stop repeatedly calling a failing dependency. They protect your system from cascading failures.	*closed/open/half-open, fallback*
Timeouts	Every network call needs a timeout. Infinite waits tie up resources and create invisible reliability problems.	*connect timeout, read timeout, deadline*
Retries	Retries help transient failures but must use limits, backoff, jitter, and idempotency. Blind retries can create outages.	*retry budget, exponential backoff*
Backpressure	Backpressure prevents producers from overwhelming consumers. Queues, streams, and APIs need load-shedding strategies.	*queue depth, rate control, load shedding*
Graceful Degradation	A system should serve partial functionality when dependencies fail. This improves user trust and reduces all-or-nothing failure.	*fallback cache, read-only mode, degraded UI*
Postmortems	Postmortems document what happened, why, impact, detection, and prevention. They should be blameless and action-oriented.	*timeline, contributing factors, action items*

13. Testing Strategy and Code Quality

Testing is not just unit tests: it is a layered confidence system across frontend, backend, DB, APIs, queues, and deployments.

Topic	Simple two-line brief	Deep-dive keywords
Unit Tests	Unit tests check isolated functions or small modules. They are fast and best for business rules, utilities, reducers, and validators.	*Jest, Vitest, pure functions*
Integration Tests	Integration tests check modules working together, often with DB/API dependencies. They catch wiring bugs unit tests miss.	*test database, API tests, containers*
E2E Tests	E2E tests simulate real user flows through the browser and backend. Use them for critical paths, not every tiny edge case.	*Playwright, Cypress, login flow*
Contract Tests	Contract tests verify that API producers and consumers agree on shape and behavior. They are valuable in service-heavy systems.	*OpenAPI, Pact, schema tests*
Regression Tests	Regression tests lock in fixes for bugs that already happened. Every serious production bug should produce a test or monitor.	*bug reproduction, fixture*
Load Testing	Load tests reveal performance under traffic. Measure p95 latency, throughput, DB saturation, queue depth, and error rates.	*k6, Artillery, autocannon*
Snapshot Tests	Snapshots catch unexpected UI/output changes but can become noisy. Use carefully for stable structures, not dynamic UI everywhere.	*snapshot review, brittle tests*
Mocking	Mocking isolates dependencies but can hide integration bugs. Mock external APIs carefully and keep critical integration coverage.	*test doubles, MSW, fake timers*
Test Data Management	Reliable tests need predictable data setup and cleanup. Factories, transactions, and isolated databases reduce flaky behavior.	*fixtures, factories, seed data*
Linting and Formatting	Linters catch code smells and unsafe patterns; formatters prevent style debates. Together they reduce review noise.	*ESLint, Prettier, typecheck*
Code Review	Good reviews check correctness, maintainability, security, tests, observability, and product edge cases. They are not just syntax comments.	*review checklist, design review*
Refactoring	Refactoring changes structure without changing behavior. Safe refactoring needs tests, small steps, and clear boundaries.	*strangler pattern, component extraction*

14. System Design Core Concepts

This is the compact full-stack system design vocabulary you should be able to explain and apply to real products.

Topic	Simple two-line brief	Deep-dive keywords
Scalability	Scalability means handling more load by adding resources or improving efficiency. Think separately about users, requests, data size, and background jobs.	*vertical scale, horizontal scale*
Availability	Availability measures whether the system can serve users when needed. It depends on redundancy, failover, isolation, and operational discipline.	*uptime, redundancy, failover*
Reliability	Reliability means correct behavior over time under expected and unexpected conditions. It includes retries, data correctness, recovery, and graceful failure.	*fault tolerance, recovery, durability*
Latency vs Throughput	Latency is time per request; throughput is total work per time. Optimizing one can hurt the other, so know which user experience matters.	*p50, p95, QPS, saturation*
Load Balancing	Load balancers distribute traffic across instances. Algorithms and health checks affect fairness, stickiness, failover, and overload behavior.	*round robin, least connections, sticky sessions*
Caching	Caching stores expensive results closer to the caller. The hard parts are invalidation, consistency, freshness, and cache stampedes.	*browser cache, CDN, Redis, DB cache*
Database Replication	Replication copies data to other nodes for reads or failover. It improves availability but introduces lag and consistency tradeoffs.	*sync/async replication, replica lag*
Partitioning / Sharding	Sharding splits data across machines. Choose shard keys carefully because bad keys create hot partitions and hard migrations.	*shard key, range/hash sharding*
Consistent Hashing	Consistent hashing distributes keys with minimal movement when nodes change. It is used in caches, distributed stores, and load distribution.	*hash ring, virtual nodes*
CAP Theorem	Under network partition, a distributed system must choose between consistency and availability. Real design is about concrete failure modes, not slogan answers.	*consistency, availability, partition tolerance*
Consistency Models	Strong consistency gives latest data; eventual consistency allows temporary divergence. Product requirements decide the acceptable model.	*read-after-write, monotonic reads*
Leader-Follower Architecture	One leader handles writes and followers replicate for reads/failover. It simplifies consistency but makes leader failure important.	*primary-replica, failover, election*
Message Queues	Queues decouple producers and consumers and smooth spikes. They introduce ordering, retry, idempotency, and visibility-timeout concerns.	*at-least-once, exactly-once illusion*
Event-Driven Architecture	Events represent facts that happened and let other parts react asynchronously. It improves decoupling but needs schema/version discipline.	*event log, consumers, event schema*
CQRS	CQRS separates write models from read models. Useful when read views differ heavily from write logic, but it adds complexity.	*commands, queries, projections*

Topic	Simple two-line brief	Deep-dive keywords
Saga Pattern	Sagas coordinate multi-step workflows without one global transaction. Each step has a compensating action for failure handling.	*orchestration, choreography, compensation*
Idempotency in Distributed Systems	Distributed systems retry, duplicate, and reorder work. Idempotency makes repeated operations safe.	*dedupe table, idempotency key, upsert*
Backpressure and Load Shedding	When overloaded, systems should slow producers or reject work intentionally. This preserves core functionality instead of collapsing.	*429, queue limits, admission control*

15. Rate Limiting Deep Dive

A concrete system design topic you asked for: algorithms, keys, tradeoffs, Redis implementation, and distributed concerns.

Topic	Simple two-line brief	Deep-dive keywords
Why Rate Limit?	Rate limiting protects APIs from abuse, accidental spikes, expensive workloads, and noisy tenants. It is both a reliability and product-policy mechanism.	*abuse prevention, quota, fairness*
Rate Limit Key	The key defines who/what is limited: IP, user ID, API key, org, route, model, or tenant. Wrong keys either block good users or fail to stop abuse.	*IP, user, org, endpoint, plan*
Fixed Window Counter	Counts requests in discrete windows like per minute. It is simple and cheap but allows bursts at window boundaries.	*INCR, EXPIRE, window boundary*
Sliding Window Log	Stores request timestamps and counts only recent ones. It is accurate but memory-heavy for high traffic.	*sorted set, timestamps, cleanup*
Sliding Window Counter	Approximates sliding behavior with current and previous windows. It is less memory-heavy than logs and smoother than fixed windows.	*weighted window, approximation*
Token Bucket	Tokens refill at a steady rate and each request spends one token. It allows controlled bursts while enforcing an average rate.	*refill rate, bucket capacity, burst*
Leaky Bucket	Requests drain at a constant rate like water from a bucket. It smooths bursts more aggressively and protects downstream systems.	*queue, constant drain, smoothing*
Concurrency Limit	Limits simultaneous in-flight work instead of requests per time window. Useful for expensive tasks, LLM calls, uploads, and build jobs.	*semaphore, in-flight count, worker slots*
Cost-Based Limit	Charges requests by cost instead of count. This is useful when one request can be much more expensive than another, like AI tokens or file processing.	*tokens, compute units, weighted requests*
Hierarchical Limit	Applies multiple limits at once: per IP, per user, per org, per endpoint, and global. This prevents one dimension from bypassing policy.	*global quota, tenant quota, endpoint quota*
Distributed Rate Limiting	Multiple app servers need shared state or approximate local limits. Redis is common, but latency and partitions affect correctness.	*Redis, local cache, AP tradeoff*
Atomicity	Rate limit check and update must be atomic under concurrency. Redis Lua scripts or single commands avoid race conditions.	*Lua script, MULTI/EXEC, race condition*
429 Response Design	A good rate-limit response includes 429 status and useful headers so clients can back off. Avoid vague failures.	*Retry-After, X-RateLimit-Remaining*
Fail Open vs Fail Closed	If the limiter store fails, you either allow traffic or block traffic. Choose based on risk: public APIs may fail closed; user apps may fail open.	*availability vs abuse risk*
Bypass and Abuse Cases	Attackers rotate IPs, users, or tokens. Combine keys, device signals, auth state, WAF, and anomaly detection for serious abuse.	*IP rotation, bot traffic, fraud*

16. Common System Design Patterns for Your Target Roles

Patterns you should be able to sketch for full-stack/product/AI-infra startups.

Topic	Simple two-line brief	Deep-dive keywords
Notification System	Design persistent notifications plus realtime delivery. Store events, fan out via workers, deliver via WebSocket/SSE, and replay missed notifications on reconnect.	*outbox, fanout, unread count, replay*
File Processing Pipeline	Uploads should go to object storage, enqueue processing jobs, track status, and notify users on completion. Avoid keeping long processing inside request handlers.	*presigned URL, queue, worker, status table*
Live Build/Preview System	A build system needs job isolation, logs, artifact storage, status streaming, timeout cleanup, and preview routing. Edward maps directly to this design.	*sandbox, logs, preview URL, cleanup*
Chat System	Chat needs message persistence, ordering, delivery receipts, connection state, fanout, and offline sync. Realtime transport is only one piece.	*WebSocket, message ID, unread, ordering*
Dashboard System	Dashboards need data ingestion, aggregation, caching, freshness indicators, alerting, and drill-down. Monitoring dashboards are production systems.	*metrics pipeline, charts, alert rules*
Search System	Search combines indexing, ranking, filters, pagination, and freshness. Hybrid lexical/semantic search is now common in AI products.	*inverted index, embeddings, rerank*
Multi-Tenant SaaS	Multi-tenancy requires tenant isolation in auth, data, rate limits, billing, logs, and admin access. Tenant leakage is a severe bug.	*tenant_id, row-level security, quotas*
Audit Log System	Audit logs record who did what, when, and from where. They need append-only storage, immutable semantics, and queryability.	*actor, action, resource, timestamp*
Webhook Delivery System	A delivery system stores events, signs payloads, retries with backoff, exposes delivery logs, and supports manual replay.	*outbox, HMAC, retry, endpoint health*
Job Queue System	A queue system needs producers, workers, retries, priority, concurrency, idempotency, DLQ, and operational visibility.	*BullMQ, worker pool, DLQ, metrics*
Rate Limiter Service	A rate limiter service checks policies atomically, returns allow/deny plus reset metadata, and supports dynamic rules per tenant/plan.	*policy store, Redis, Lua, headers*
URL Shortener	A classic design for IDs, redirects, cache, analytics, abuse prevention, and high-read traffic. Good for practicing read-heavy architecture.	*base62, cache, redirect, analytics*
Feed System	Feeds need ranking, fanout-on-write vs fanout-on-read, pagination, cache, and personalization. Choose based on follower graph size and freshness needs.	*timeline, fanout, ranking, cursor*
Feature Flag System	Feature flags need config storage, evaluation rules, SDK caching, rollout percentages, audit logs, and kill switches.	*targeting, percentage rollout, SDK cache*
Analytics/Event Tracking System	Event systems need SDKs, ingestion API, schema validation, queue/buffer, storage, aggregation, and privacy handling. Your Iterate work maps here.	*events, batching, schema registry, ETL*

17. Performance Engineering Across the Stack

Performance is end-to-end: frontend load, API latency, DB queries, cache behavior, network, and background work.

Topic	Simple two-line brief	Deep-dive keywords
Frontend Bundle Size	Large JS bundles slow first load and hydration. Reduce through code splitting, tree shaking, dynamic imports, and removing unused dependencies.	*bundle analyzer, dynamic import, tree shaking*
Core Web Vitals	Vitals measure real user experience: loading speed, responsiveness, and visual stability. They are useful but must be paired with product-specific metrics.	*LCP, INP, CLS*
Image Performance	Images often dominate page weight. Use responsive sizes, modern formats, lazy loading, CDN, and correct priority.	*WebP, AVIF, srcset, lazy loading*
API Latency	API latency includes network, app processing, DB time, external calls, serialization, and queue waits. Trace before optimizing.	*p95, tracing, flamegraph*
N+1 Queries	N+1 happens when code makes one query per item. Fix with joins, batching, eager loading, or DataLoader-style batching.	*joins, include, DataLoader*
Query Optimization	Use EXPLAIN, indexes, query shape changes, and data modeling to optimize DB queries. Guessing is weaker than reading plans.	*EXPLAIN ANALYZE, index scan, sort*
Connection Saturation	Too many app instances can overwhelm DB connections. Pooling, PgBouncer, and queueing protect the DB.	*pool size, max connections, saturation*
Compression	Compressing responses reduces bandwidth but uses CPU. Use gzip/brotli where appropriate and avoid compressing already-compressed data.	*gzip, brotli, content-encoding*
Batching	Batching combines small operations to reduce overhead. Useful for API calls, DB writes, analytics events, and queue jobs.	*bulk insert, request batching, flush interval*
Debounce and Throttle	Debounce waits for quiet; throttle limits frequency. Use them for search input, resize, scroll, and expensive client actions.	*debounce search, throttle scroll*
Streaming	Streaming improves perceived latency by sending partial results early. Useful for AI responses, large pages, and long-running progress.	*SSE, chunks, backpressure*
Profiling	Profiling shows where time or memory is actually spent. Use browser profiler, Node profiler, DB explain, and production traces.	*CPU profile, heap snapshot, trace*

18. Product Engineering and Maintainability

This is the engineering judgment layer: tradeoffs, clarity, release safety, user impact, and codebases that survive growth.

Topic	Simple two-line brief	Deep-dive keywords
Domain Modeling	Domain modeling names the real concepts in the product. Good models reduce accidental complexity and make APIs easier to understand.	*entities, value objects, invariants*
Separation of Concerns	Keep UI, business logic, data access, and infrastructure boundaries clear. It makes code easier to test, replace, and debug.	*controllers, services, repositories*
Modular Architecture	Modules should own related behavior and hide internals. Avoid giant shared folders that become dumping grounds.	*feature modules, bounded context*
Tech Debt	Tech debt is a conscious shortcut with future cost. Mature engineers identify, isolate, and pay it down when it blocks velocity.	*debt register, refactor window*
Build vs Buy	Build when differentiation/control matters; buy when commodity reliability matters. This applies to auth, search, analytics, queues, and infra.	*vendor risk, lock-in, time-to-market*
MVP vs Scale	Early systems should be simple but not careless. Design for clear upgrade paths instead of premature microservices.	*monolith, modular monolith, migration path*
Operational Readiness	A feature is not production-ready until it has monitoring, failure handling, rollback, and support visibility. Shipping code is not the finish line.	*runbook, dashboard, alert, rollback*
Documentation	Docs should explain decisions, setup, APIs, failure modes, and ownership. Good docs reduce team dependency and onboarding time.	*README, ADR, API docs, runbook*
Architecture Decision Records	ADRs capture important decisions and tradeoffs. They help future engineers understand why the system is the way it is.	*context, decision, consequences*
User-Centric Debugging	Debug from user symptom to system cause. This keeps engineering tied to product impact instead of only internal correctness.	*repro steps, impact, severity*

19. 30-day full-stack revision plan

This plan is designed for someone who can already build full-stack apps, but wants stronger fundamentals, vocabulary, and system design confidence.

Timebox	Focus
Week 1	JS/TS runtime, browser/HTTP, React rendering, hooks, state, TanStack Query/Zustand, frontend perf.
Week 2	Next.js app architecture, Node/Express, API design, auth, validation, errors, WebSockets/SSE/webhooks.
Week 3	PostgreSQL, MongoDB, Redis, BullMQ, caching, queues, transactions, indexes, query planning.
Week 4	Docker, AWS basics, deployment, observability, testing, security, system design and rate limiting deep dive.

Daily structure

- 45 minutes: read one core topic from this map and write your own explanation in 6-8 lines.
- 45 minutes: build or inspect a tiny implementation example related to the topic.
- 30 minutes: answer 5 interview-style questions out loud and refine weak wording.
- Weekly: take one project from your resume and re-explain it using the week's concepts.

Mini-labs to build while revising

Topic	Simple two-line brief	Deep-dive keywords
Rate limiter lab	Implement fixed window, sliding window log, token bucket, and concurrency limiter using Redis. Add 429 headers and tests.	*implementation, tests, notes*
Query planner lab	Create Postgres tables, add indexes, run EXPLAIN ANALYZE before/after, and document what changed.	*implementation, tests, notes*
Queue reliability lab	Build a BullMQ worker with retries, exponential backoff, DLQ, idempotency keys, and a small dashboard.	*implementation, tests, notes*
Next.js caching lab	Build the same page using static, dynamic, revalidated, streamed, and client-fetched patterns. Compare behavior.	*implementation, tests, notes*
Observability lab	Add request IDs, structured logs, metrics, traces, and dashboard panels for one small API + worker system.	*implementation, tests, notes*

20. Interview checkpoints

These are the answers you should be able to say naturally, without sounding memorized.

Topic	Simple two-line brief	Deep-dive keywords
Explain your stack in one senior-sounding paragraph	I build full-stack AI/product systems using TypeScript, React/Next, Node/Express, Postgres/Redis, Docker, queues, and realtime streams, with focus on reliability, recovery, and product UX.	*interview wording, project tie-in*
Explain Edward from full-stack angle	Edward is a job-driven app-generation platform: Next frontend, Express APIs, BullMQ workers, Docker sandboxes, Redis-buffered writes, Postgres state, S3 artifacts, and preview routing.	*interview wording, project tie-in*
Explain Agentic Chat from full-stack angle	Agentic Chat combines Next.js UI, Postgres/pgvector persistence, async document processing, tool orchestration, caching, security controls, and long-running state checkpoints.	*interview wording, project tie-in*
Explain a production bug you can debug	Trace the symptom through UI, API logs, request IDs, DB query plans, queue state, cache keys, worker retries, and deployment changes.	*interview wording, project tie-in*
Explain frontend performance	Measure first, then reduce JS, fix render waterfalls, optimize images, use caching/streaming, memoize carefully, and track real user vitals.	*interview wording, project tie-in*
Explain backend reliability	Use timeouts, retries with backoff, idempotency, queues, DLQs, circuit breakers, health checks, logs, metrics, traces, and rollback paths.	*interview wording, project tie-in*
Explain rate limiting	Pick a key, choose algorithm based on burst tolerance and accuracy, implement atomic check/update in Redis, return 429 headers, and define fail-open/fail-closed behavior.	*interview wording, project tie-in*

21. Research prompts for agents/partners

Use these prompts with research-oriented agents, mentors, or study partners. Ask for examples, diagrams, interview questions, and mini-projects for each.

1. Explain the JavaScript event loop in Node.js with microtasks, macrotasks, nextTick, setImmediate, timers, and I/O phases. Include interview examples and debugging symptoms.
2. Create a deep guide to React rendering: render vs commit phase, reconciliation, keys, hooks, stale closures, memoization, context performance, and memory leaks.
3. Explain Next.js App Router deeply: Server Components, Client Components, caching layers, streaming, route handlers, middleware, server actions, and deployment tradeoffs.
4. Create a complete Node.js/Express backend checklist: routing, middleware, validation, auth, error handling, graceful shutdown, config, logging, testing, and deployment.
5. Explain PostgreSQL indexes and query plans with examples: B-tree, composite, partial, GIN, EXPLAIN ANALYZE, locks, transactions, isolation levels, and migrations.
6. Design a Redis-backed rate limiter covering fixed window, sliding log, sliding counter, token bucket, leaky bucket, concurrency limit, Lua atomicity, and distributed failure modes.
7. Design a production job queue system using BullMQ/Redis: states, retries, backoff, idempotency, DLQ, workers, concurrency, monitoring, and recovery.
8. Explain full-stack security for a Next.js/Node app: cookies, JWT, OAuth, CSRF, XSS, SQLi, SSRF, CORS, rate limits, secrets, and security headers.
9. Create a system design for Edward-like app builder: API layer, job queue, Docker sandbox isolation, preview routing, logs, artifact storage, recovery, cleanup, and monitoring.
10. Create a study plan to go from 1-2 YOE full-stack to strong 4-5 YOE interview depth across TypeScript, React, Next, Node, DBs, Redis, Docker, AWS, testing, and system design.

22. Source trail and deep-reading map

Use official docs and primary references first, then blogs/videos. This prevents shallow tutorial-based understanding.

Source	Use it for
React official docs	Components, hooks, state, effects, rendering mental model.
Next.js official docs	App Router, data fetching, caching, rendering, route handlers, middleware.
MDN Web Docs	JavaScript execution model, HTTP, CORS, status codes, caching, browser APIs.
Node.js official docs	Event loop, async work, timers, nextTick, non-blocking I/O.
Express official docs	Routing, middleware, request/response lifecycle.
TanStack Query docs	Server state, caching, invalidation, mutations, hydration.
Zustand docs	Client state stores, selectors, hooks-based state management.
PostgreSQL official docs	Indexes, EXPLAIN, transactions, isolation, locks, replication, backups.
Redis docs	Data structures, caching, rate limiting, streams, counters, TTLs.
Docker docs	Images, containers, networks, volumes, Compose, build strategies.
AWS Well-Architected Framework	Operational excellence, security, reliability, performance, cost, sustainability.
OAuth 2.0 RFC 6749	Authorization flows, access tokens, refresh tokens, scope, clients.

Final self-positioning line

Use this in interviews

I am strongest at building full-stack AI/product systems end to end: React/Next frontends, Node APIs, Postgres/Redis data layers, queues/workers, Docker/AWS deployment, realtime streams, monitoring, and production reliability. I am revising fundamentals deeply so I can explain not just what I built, but why each system design choice works.