System Design Guide

APIs, Databases, Caching, Load Balancing & Production Infrastructure

---

Prepared For: Enterprise IT Professionals & System Architects

Course Author: Darien Buchanan

Format: Step-by-Step Chronological Architectural Guide

Date: June 2026

Chapter 1: Foundations & The Single Server Setup

Every massive, distributed web topology supporting millions of users originates from a simple bedrock configuration. To design high-performance architectures, engineers must first master the fundamental request-response loop and identify structural bottlenecks within a localized, single-instance ecosystem.

1.1 Architectural Concept

In a baseline setup, all core sub-systems—the business logic layer (web application), data layer (database), and localized memory caching—reside co-located on a solitary virtual or physical server bound to a public-facing IP address. Traffic flows directly from client applications to this single point of compute.

Step-by-Step Request Execution Procedure

1. **Domain Name Resolution:** The user inputs a human-readable domain name (e.g., `app.demo.com`) via a mobile application or standard browser layer. The client system dispatches an upstream request to the Domain Name System (DNS).
2. **IP Mapping Response:** The DNS provider references internal network zone configurations to resolve and map the text domain to the static, target server internet protocol location (e.g., public IPv4 address).
3. **Network Handshake & Protocol Ingestion:** Armed with the raw IP destination, the client platform transmits an explicit HTTP network request across standard transport sockets to the destination engine.
4. **Application Processing:** The single server acts as an all-in-one resource container, dynamically rendering static assets (HTML/CSS/JS) for browsers or packaging structured raw JSON database objects for mobile applications.

1.2 Required Tools & Command Frameworks

- **DNS Utilities:** `nslookup`, `dig` (for querying Domain Name System mapping records directly via the command-line network interfaces).
- **Client Interrogation:** Web Browsers, Mobile Platforms, or raw HTTP runtimes (e.g., `curl`, Postman clients).

Chapter 1 Best Practices

- **Isolate Traffic Types:** Establish a clear programmatic separation early in development by using distinct presentation formats. Serve fully integrated markup blocks exclusively for web browsers, and light structural text nodes (clean JSON structures) for mobile application endpoints.
- **Evaluate Growth Constraints:** Constantly log resource utilisation (CPU core spikes, memory depletion, storage I/O limits) to determine precisely when a single server setup must transition to an enterprise tier.

Chapter 2: Database Tiering & Selection Strategy

As standard application demand steps outside a limited, small-scale configuration, the data persistence layer must be extracted entirely from the compute node, transforming the architecture into distinct web/mobile application tiers and dedicated database tiers.

2.1 Core Paradigm Matrix

System designers primarily choose between Relational Database Management Systems (RDBMS) governed by strict schemas, or Non-Relational (NoSQL) frameworks built for unstructured and high-velocity horizontal scaling patterns.

Database Model	Core Architecture / Engine Examples	Primary System Strengths	Optimal Enterprise Use Cases
Relational (SQL)	PostgreSQL, MySQL, Oracle, SQLite	Strict schemas, tabular row/column data storage, complex cross-table relational JOIN operators, strict transaction control.	E-commerce order mapping systems, transactional financial systems, accounting journals.
Document Store	MongoDB	Flexible, schema-less nested JSON-like document entities, complex nested structural models packaged inside single files.	Content Management Systems, dynamic application metadata, catalog management.
Wide - Column Store	Cassandra, Cosmos DB	Tabular layouts utilizing dynamic, changing structural columns; heavily optimized for extreme multi-node physical write scalability.	Massive timeseries logging, industrial IoT telemetry capture, telemetry streams.
Key-Value Store	Redis, Memcached	Simplicity, ultra-low latency index mappings kept directly inside volatility RAM for sub-millisecond retrieval patterns.	Distributed session persistence, heavy global data caching layers.
Graph Database	Neo4j, Amazon Neptune	Native storage of graph networks containing interconnected complex multi-axis nodes and edge relationship configurations.	Social networking engines, real-time product/behavior recommendation machines.

Step-by-Step Data Tier Separation Procedure

1. **Provision Target Database Compute:** Launch a separate, high-performance database cluster entirely isolated from the main application logic instances.
2. **Enforce Transactional Consistency (ACID Execution):** For relational dependencies, implement full transaction patterns following the ACID acronym:
    - *Atomicity:* Ensure a complete sequence of SQL writes operates as a single unit—either succeeding perfectly or rolling back fully.
    - *Consistency:* Validate that modifications push data strictly from one valid structural state to another.
    - *Isolation:* Concurrently executing processes must be cleanly isolated, avoiding data intersections.
    - *Durability:* Commit completed state data into non-volatile storage blocks resistant to hardware failures.
3. **Embed Structured Payload Architectures:** For NoSQL targets (such as MongoDB), bundle associated relational components (e.g., structural customer logs, historical product links, metadata maps) inside a single, holistic nested JSON document object to completely avoid cross-node network lookups.

Chapter 2 Best Practices

- **Map Systems to Data Models:** Deploy Relational/SQL engines whenever schemas are predictably structured and require complete transactional security. Transition to NoSQL options when processing massive, changing, unstructured, or semi-structured high-volume data payloads.
- **Minimize Relational Cross-Joins:** Avoid nesting multi-table SQL queries deeply over heavy datasets to reduce database memory locking and protect query runtimes.

Chapter 3: Horizontal vs. Vertical Scaling & Load Balancing

A growing system eventually runs into resource caps on individual hardware engines, requiring a architectural pivot from basic raw hardware additions to advanced scale-out structures.

3.1 Core Scaling Approaches

- **Vertical Scaling (Scale Up):** Adding raw system resources (more physical RAM, higher CPU core allocations, expanded storage speeds) to a single machine server frame.
  Constraints: Hard physiological hardware threshold ceilings; complete lack of system redundancy (single point of failure).
- **Horizontal Scaling (Scale Out):** Deploying multiple, identical application servers in parallel behind a single routing point to evenly distribute structural runtime traffic load.
  Strengths: Unlimited scale-out ceiling, excellent operational fault tolerance.

3.2 Load Balancer Algorithm Matrix

To safely bridge user connections across horizontally scaled application servers, engineers use central load balancers that distribute incoming requests using specific routing rules.

Routing Algorithm	Technical Operational Logic	Ideal System Infrastructure Application
Round Robin	Directs traffic sequentially across a circular, rotating list of healthy backend servers.	Environments with homogeneous backend hardware capacities and uniform request loads.
Least Connections	Evaluates active server sessions and pushes new traffic to the node processing the lowest concurrent connections.	Applications running sessions of variable length (e.g., persistent real-time portals).
Least Response Time	Monitors absolute server latency and routes requests to the fastest, most responsive node with the fewest active sessions.	Heterogeneous infrastructure clusters where response speeds fluctuate between compute instances.
IP Hash	Transforms the caller's unique IP address into a hash index to stick that user to a specific backend machine.	Legacy stateful server applications requiring session persistence without centralized caches.
Weighted Algorithms	Assigns customized traffic ratios to servers based on hardware specs (e.g., RAM/CPU ratings).	Mixed cloud environments combining legacy servers alongside new high-performance machines.
Geographical Routing	Inspects client destination source country data via IP lookups and sends them to the nearest server region.	Global applications where reducing regional network latency is critical for user experience.
Consistent Hashing	Maps nodes and users to a shared mathematical hash ring (Si), routing requests to the closest node.	Distributed caching systems and scalable storage rings to minimize re-indexing during scaling events.

Step-by-Step Horizontal Scaling Setup

1. **Replicate Backend Application State:** Replicate structural web application builds identically across multiple isolated compute nodes (e.g., Server 1, Server 2, Server 3).
2. **Position Load Balancing Intermediary:** Inject an enterprise load balancer proxy engine as the primary interface layer between client callers and the compute pool.
3. **Configure Heartbeat Health Checks:** Set up continuous health monitoring requests to track cluster status. The load balancer pings target backend nodes via custom interval requests; if a node drops offline or fails health assertions, the proxy pulls it from the active rotation to protect incoming user traffic.

3.3 Required Tools & Configuration Systems

- **Software Load Balancers / Reverse Proxies:** Nginx (all-in-one web platform and routing gateway), HAProxy (high-performance open-source load balancer).
- **Hardware Infrastructure Controllers:** F5 BIG-IP, Citrix ADC engines.
- **Managed Cloud Balancers:** AWS Elastic Load Balancing (ELB), Azure Load Balancer, Google Cloud Load Balancing ecosystems (supporting automatic scaling and integrated traffic monitoring out of the box).

Chapter 3 Best Practices

- **Enforce State Isolation:** Ensure application servers are completely stateless. Extract local user sessions, file shares, and variables out of volatile machine memory and move them to centralized distributed caches or shared datastores.
- **Design Multi-Tier Redundancy:** Avoid a Single Point of Failure (SPOF). Deploy load balancers in redundant, active-passive pairs managed by automated tracking scripts. This allows a backup load balancer to take over instantly if the primary router encounters an unexpected failure.

Chapter 4: API Paradigm Design & Architectural Protocol Selection

API components establish the contractual boundaries and interaction patterns across software systems. Selecting the correct protocol heavily influences system responsiveness, operational data overhead, and developer velocity.

4.1 Core Architecture Typology

- **REST (Representational State Transfer):** Resource-centric structural architectural pattern leaning heavily on native HTTP verbs, resource-focused URLs, stateless operations, and distinct semantic response metadata codes. Highly compatible across the web and mobile spaces.
- **GraphQL:** Client-driven data graph query runtime framework utilizing an explicit structural schema contract layout. Allows client apps to specify precise payload field requirements over a single endpoint request block.
- **gRPC (Remote Procedure Call):** Extremely efficient, high-performance transport system created by Google. Combines HTTP/2 streaming transport pipelines alongside structured binary protocol buffer payloads.

Architectural Attribute	RESTful API Paradigms	GraphQL Language Frameworks	gRPC Transport Frameworks
Endpoint Structure	Resource-based plural nouns (e.g., `/v1/products`).	Single central entrypoint (e.g., `/graphql`).	Remote functions defined inside `.proto` files.
Data Payload Control	Fixed schema structures (risks over/under-fetching).	Client-selected parameters (eliminates over-fetching).	Serialized binary format buffers (ultra-compact).
Version Management	Explicit URI segmentation (e.g., `/v1`, `/v2`).	Continuous evolution without version prefixes.	Protocol Buffer field number markers.
Caching Capabilities	Native HTTP-level network caching rules.	Central application-layer optimization graphs.	Internal point-to-point server handling.
N et work Optimization	Requires multiple target requests for nested resources.	Single request fetches complete nested relational graphs.	Multiplexed binary pipelines via HTTP/2.

Step-by-Step RESTful Resource Modeling Procedure

1. **Translate Domain Models to Plural Nouns:** Convert core application domains into clean plural resource identifiers rather than functional verbs. Map operational endpoints using patterns such as `/v1/products` or `/v1/orders`. Do not use verb-driven routes like `/getProducts`.
2. **Map Clean CRUD Operations to HTTP Verbs:**
   - `GET`: Read collection arrays or isolated data elements (must operate cleanly as safe and idempotent).
   - `POST`: Create brand-new target entities inside the datastore.
   - `PUT`: Fully replace an existing data resource with an entirely new incoming record payload.
   - `PATCH`: Partially alter specific internal attributes of an existing object without overwriting unchanged fields.
   - `DELETE`: Purge specified data targets cleanly from the underlying database storage layers.
3. **Implement Query Parameters for Performance:** Build precise data navigation tools into your `GET` collection endpoints:
   - *Filtering:* `GET /v1/products?category=electronics&inStock=true`
   - *Sorting:* `GET /v1/products?sort=price_asc`
   - *Pagination:* Integrate `page`/`limit` schemas, `offset` counters, or high-performance cursor hashes (e.g., `?cursor=xyz&limit=10`) to prevent server memory saturation during mass exports.
4. **Standardise Global HTTP Status Returns:**
   - `200 OK`: Resource query or update executed successfully.
   - `201 Created`: New entity successfully added to the database.
   - `400 Bad Request`: Malformed structural data input block captured from the client app.
   - `401 Unauthorized`: Valid identity verification credentials missing from call headers.
   - `404 Not Found`: Searched data item missing from persistence systems.
   - `500 Internal Server Error`: Core application runtime crash encountered on the backend server.

Chapter 4 Best Practices

- **Enforce Explicit REST API Versioning:** Always isolate code builds using clear API pathway identifiers (e.g., /api/v1/...). This prevents backward-breaking interface updates from disrupting deployed mobile and web client platforms.
- **Mitigate GraphQL Depth Abuse:** Implement strict deep-query analyzers and limit depth constraints (e.g., capping query execution at a maximum of 6–7 nested layers). This blocks complex recursive queries from causing memory depletion on backend server resources.
- **Optimize Microservice Performance via gRPC:** Limit gRPC deployments primarily to private internal datacenter networks and microservice communications. This architecture takes full advantage of compressed binary serialization and streaming capabilities.

Chapter 5: Network Protocols & Transport Layer Mechanics

Application interfaces rely entirely on foundational low-level transport mechanisms. Selecting and tuning the correct transport framework is critical for maintaining data integrity and minimizing system communication latency.

5.1 Network OSI Layer Stack Positioning

Modern system communication separates operations into distinct layers. Application-layer protocols (HTTP, HTTPS, WebSockets, AMQP) define the message patterns and structures. These run on top of Transport-layer routing protocols (TCP, UDP), which manage the physical movement of data packets across network infrastructure.

5.2 Transport Protocols Selection Comparison

- **TCP (Transmission Control Protocol):** Connection-oriented, highly reliable stream delivery transport engine.
  Mechanics: Performs a mandatory 3-way handshake (SYN → SYN-ACK → ACK) to establish communication channels. It guarantees data delivery by auto-resending lost packets and ensuring correct order delivery via internal segment indexing numbers.
  Use Cases: Financial transaction portals, authentication systems, email exchanges, and core application data storage flows.
- **UDP (User Datagram Protocol):** Connectionless, lightweight, ultra-high-speed, but inherently unreliable transport engine.
  Mechanics: Fires packet arrays toward destination paths without establishing connections or using data-tracking receipts. Dropped packets are ignored to maintain raw transmission speed.

Use Cases: Real-time video conferencing, multiplayer online gaming sync frames, and live streaming video feeds.

Step-by-Step Real-Time Bi-Directional Network Optimization Procedure

1. **Evaluate Communication Patterns:** Identify endpoints where short polling models over standard HTTP generate empty responses, wasting bandwidth and spiking backend resource costs.
2. **Implement WebSocket Handshakes for Real-Time Needs:** Move live interactions (e.g., chat applications, collaborative workspaces) onto dedicated WebSocket protocol links. Execute an initial HTTP upgrade request handshake to establish a persistent, permanent, bi-directional socket pipe.
3. **Leverage Independent Push Pipelines:** Use the established WebSocket connection to let the server independently push system events directly to clients the millisecond they occur. This eliminates the latency and overhead of traditional polling models.
4. **Deploy Asynchronous Messaging via AMQP:** For decouple-and-forget backend jobs, implement an Advanced Message Queuing Protocol broker (such as RabbitMQ). Route processing jobs onto isolated queues so busy consumer workers can safely process heavy write tasks at their own speed.

Chapter 5 Best Practices

- **Enforce Global HTTPS Security Standards:** Secure all transactional web connections behind mandatory HTTPS layers using TLS/SSL data encryption. This protects application payloads in transit and safeguards data against man-in-the-middle exploits.
- **Align Protocols with Acceptable Data Loss Thresholds:** Deploy TCP whenever zero data loss is a firm business requirement. Switch to UDP or WebRTC frameworks only when transmission speed is critical and minor packet loss does not break the core user experience.

Chapter 6: Decentralized Identity, Authentication & Authorization

Safeguarding applications requires a clear operational separation between confirming a user's identity and validating their resource permission scopes.

6.1 Key Architectural Definitions

- **Authentication (AuthN):** Verifies identity claims to answer the question: *"Who is the user or calling system making the request?"*
- **Authorization (AuthZ):** Asserts explicit rule constraints to answer the question: *"What resources or actions can this authenticated identity access or execute?"*

Step-by-Step Stateless Token Authentication Flow

1. **Transmit Credential Arrays:** The client application passes raw login data securely over an encrypted HTTPS connection to the central Identity Provider.
2. **Verify State Data and Sign Claims:** The identity validation server verifies the incoming login credentials. Instead of opening a stateful session database log, it generates a stateless JSON Web Token (JWT). The server packages identity claims (such as `userId`, assigned `roles`, and token `expiry`) into the payload and signs it using a secure private key.
3. **Distribute Two-Token Configurations:** Return an interconnected token pair to the calling client application:
   - *Access Token:* Extremely short-lived token (valid for 15 minutes to 1 hour) included in the HTTP `Authorization: Bearer <Token>` header for active API queries.
   - *Refresh Token:* Long-lived token (valid for days or weeks) stored securely in an `HttpOnly`, `Secure` cookie wrapper to handle automatic access token renewal without forcing users to re-login.
4. **Validate Locally on Application Nodes:** When an API request comes in, backend microservices read the bearer token and verify its signature locally using the identity provider's public key. This authorizes the request instantly without requiring external database lookups.

6.2 Enterprise Authorization Matrix

Once identity is confirmed, the system enforces access control boundaries using an explicit authorization model tailored to the business domain.

Access Model	Functional Operational Concept	Target Production Deployment Env
Role -Based Access Control (RBAC)	Binds explicit permissions to abstract organization roles (e.g., Admin, Editor, Viewer) and maps users directly to those roles.	Content Management Systems, standard corporate back-office utilities, SaaS application portals.
Attribute-Based Access Control (ABAC)	Dynamically evaluates a combination of user attributes, resource data types, and live environment conditions (e.g., time, IP zone).	Highly restricted financial platforms, military security spaces, distributed medical records engines.
Access Control Lists (ACL)	Attaches specific permission matrices directly to individual data files, specifying exactly what actions unique users can run on that resource.	Collaborative document spaces (e.g., Google Drive sharing models, distributed cloud object file shares).

Chapter 6 Best Practices

- **Isolate Tokens from Local Storage Exposure:** Never save highly sensitive access or refresh tokens inside vulnerable browser localStorage locations. This protects tokens from being extracted via Cross-Site Scripting (XSS) injection paths.
- **Decouple Tokens from Authorization Models:** Use JWT containers solely as a structural mechanism to safely pass identity claims. Keep the core authorization business logic isolated within your backend code services.
- **Modernize Corporate Portals via OpenID Connect:** Migrate legacy XML-based SAML enterprise engines over to modern OpenID Connect (OIDC) authentication frameworks built on top of robust OAuth 2.0 authorization architectures.

Chapter 7: Edge Protection & API Vulnerability Mitigation

Public-facing APIs are constant targets for malicious exploits. Securing the perimeter requires a multi-layered defensive strategy implemented at the edge of your infrastructure.

Step-by-Step API Perimeter Defense Implementation

1. **Deploy Layered Rate Limiting Mechanisms:** Block brute-force attacks and resource abuse by enforcing multi-tiered request limits:
    - *Endpoint Level:* Apply strict request limits on resource-heavy routes (e.g., maximum 5 calls/minute on the `/v1/comments` submission engine).
    - *User/IP Level:* Cap requests from individual user accounts or unique client IP addresses to prevent automated scraping.
    - *Global System Level:* Set a high-threshold system limit across edge routing nodes to absorb mass DDoS traffic spikes and protect core availability.

2. **Implement Strict Parameterization on Ingestion Pipelines:** Block SQL and NoSQL injection attempts by completely banning raw string concatenation inside database access components. Force all incoming text through object-relational mapping (ORM) engines or parameterized query parameters.
3. **Activate an Edge Web Application Firewall (WAF):** Ingest incoming web traffic through an advanced firewall proxy layer (such as AWS WAF). Configure the firewall to analyze requests and automatically drop payloads displaying suspicious SQL syntax patterns, unmapped request methods, or malicious fuzzing headers.
4. **Isolate Administration Dashboards via Cryptographic VPN Zones:** Remove all internal admin interfaces and back-office tooling APIs from the public internet entirely. Require employees to authenticate through a secure Virtual Private Network (VPN) before they can route traffic to internal ser vices.
5. **Enforce Safe Cookie Security and Form Tokens:** Block Cross-Site Request Forgery (CSRF) exploits on cookie-authenticated session backends by requiring cryptographic CSRF verification tokens on every state-changing request. Ensure all session cookies use the `SameSite=Strict` configuration attribute.

Chapter 7 Best Practices

- **Enforce Whitelisted CORS Profiles:** Explicitly reject cross-origin wildcards (`Access-Control-Allow-Origin: *`) on private API backends. Lock down your Cross-Origin Resource Sharing (CORS) configurations to accept incoming web calls exclusively from trusted corporate domains.
- **Sanitize Input to Block XSS Injections:** Thoroughly escape and scrub all user-submitted text fields before saving them to the database. This ensures malicious scripts cannot be saved and executed later inside other users' browsers.