Cloud Application Design Patterns

DA 410/510

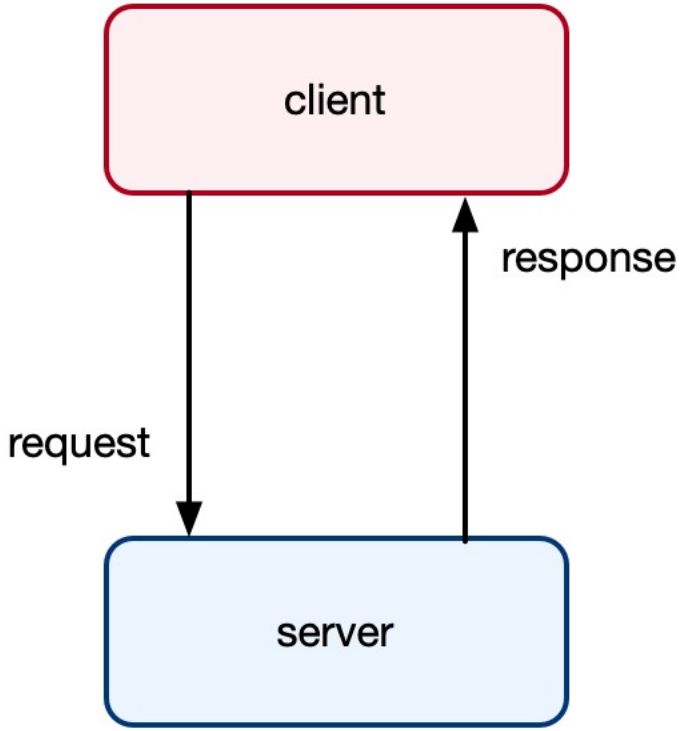
Richard Kelley

Today

- Patterns in cloud-based applications
  - three-tier architecture
  - Horizontal vs. vertical scaling
  - Load balancing
  - Replication
  - Caches and CDNs
  - Stateful vs Stateless architecture

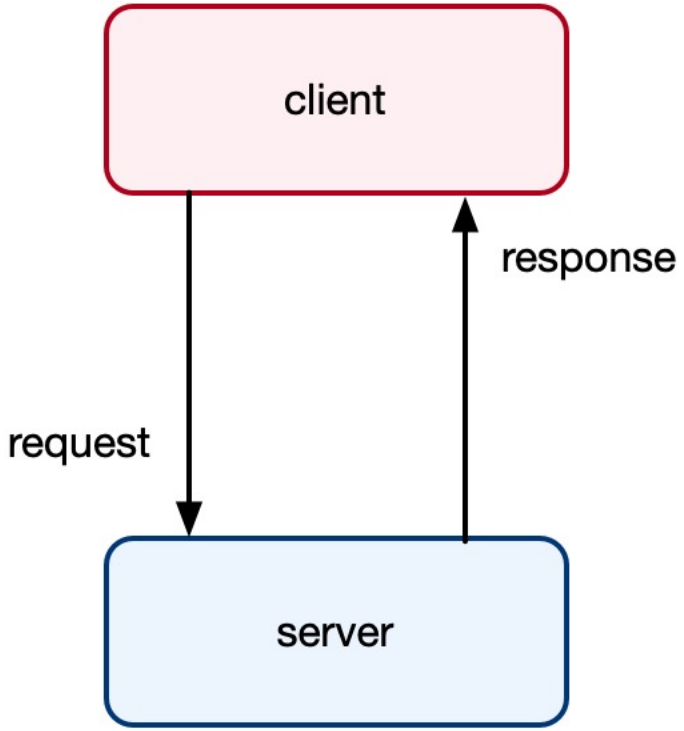
One Server

- The basic model that we've been using so far.
- The ***client*** (which is a process on a computer) sends requests to a ***server*** (which is a different process usually on a different computer).
- Activity is always initiated by the client.



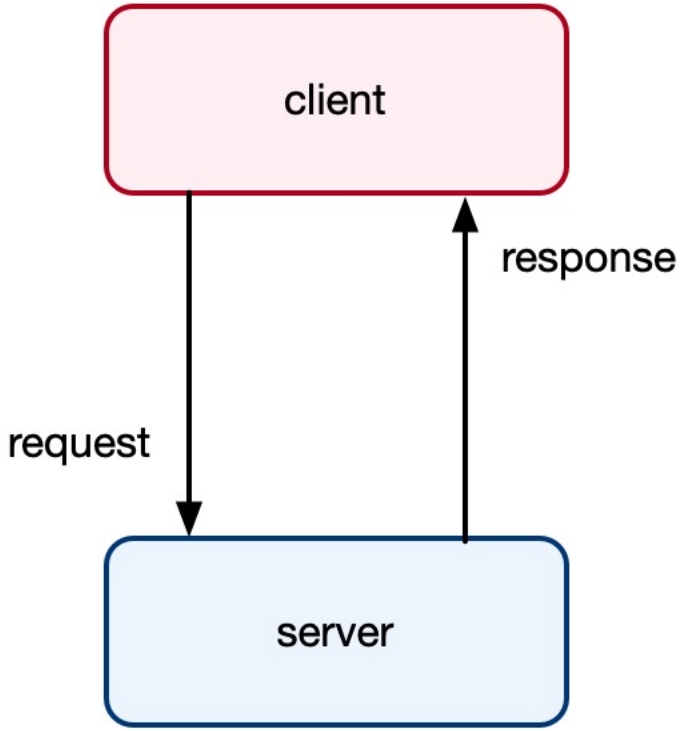
Clients

- Many programs are implemented as clients for remote servers:
  - Web browsers (the canonical example)
  - Chat clients (Discord)
  - Videogame Clients (Steam)
- ***Electron*** is a framework for using web technologies to create desktop apps
  - Codex & Claude Code are pretty good at creating electron apps.
  - If a system has a “web version” and a “desktop version” it’s probably an electron app.
  - Electron apps don’t have to be clients (VS Code), but they often are.



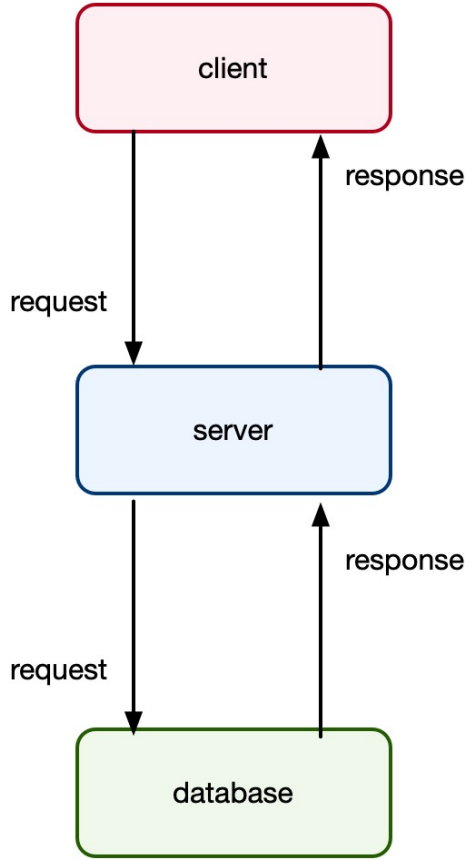
Servers

- There are many kinds of servers for many kinds of applications:
  - **nginx** is a web server.
- Lots of applications have servers.
  - Game servers (Minecraft)
  - Chat servers (XMPP/jabber)
  - Streaming servers for audio and video.



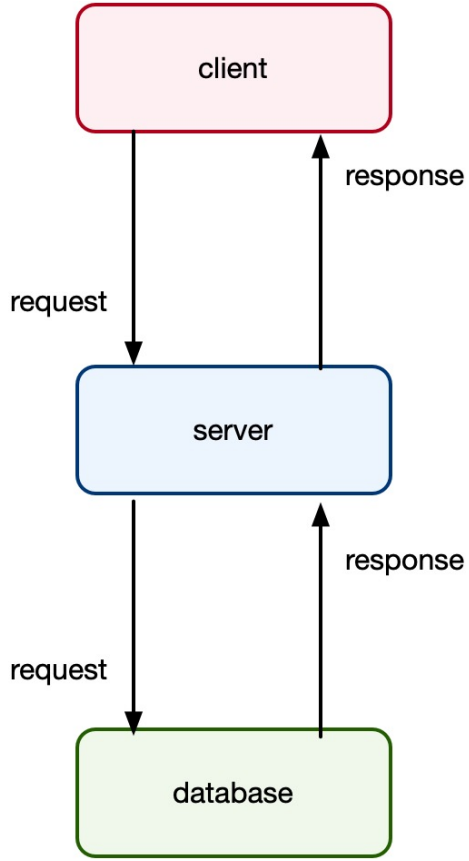
Database-Backed Applications

- It's common to store information in ***databases*** of some kind (hopefully not news).
- As "systems software," databases have grown in complexity and power, and (usually) run separately from servers.
  - Notable exception: sqlite
- It often happens that databases run as separate processes on separate machines.



Three-Tier Architecture

- ***Presentation Tier***
  - Present information to the end user.
  - Send input from the end user.
  - In web applications, this is usually client-side, and will be based on technologies like React.
- ***Application Tier***
  - Typically server-side code.
  - Contains business logic.
  - Sits between end users and data tier.
- ***Data Tier***
  - Maintains state of system



Scaling: Horizontal vs. Vertical

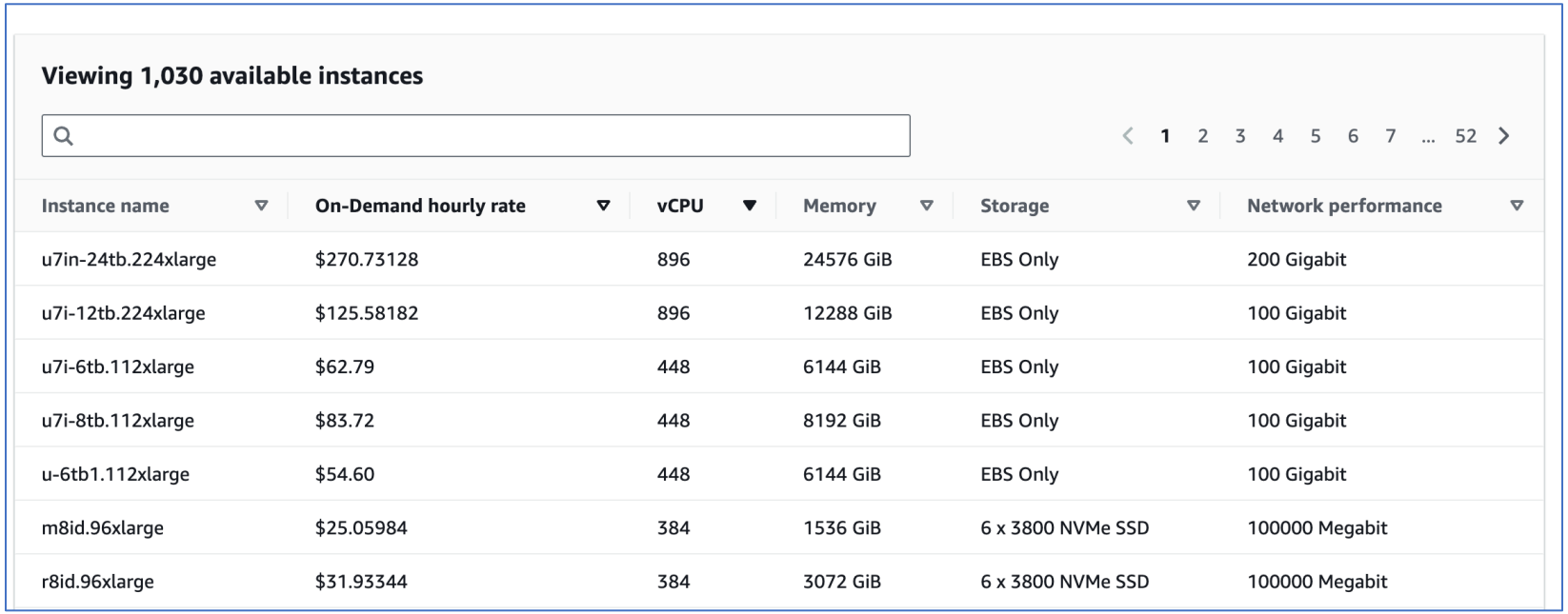
- Hypothetical: You create a website that is immensely popular. Question: How many users can you serve?
  - Suppose you run a single server on a single computer that you own in a data center you can access. What do you need to think about to figure out how many users you can support?

Scaling

- Some factors:
  - How big is the computer?
    - How long does it take to process a single request?
      - Ex: If it takes 10ms to process a request, then each core of the computer can process 100 requests/second. If your computer has 8 cores, that's about 800 requests/second.
    - How much **memory** does a response require?
      - TCP buffers, application objects, request state all require memory.
    - How much storage does a response require?
      - This is a question of space and IOPS.
  - How is the computer connected to the internet?
    - Key question here is ***Bandwidth.***
    - A 1 Gbps (gigabits/second) uplink implies 125 MB/s can be sent.
    - Ex: If the average response is 100 KB, that implies 1250 responses/second.
  - All of the above are hard limits based on the machine you're on.

Vertical Scaling

- Buy (or rent) a bigger machine.
- There are limits, but on AWS you can get big machines.
- m8id.96xlarge (general purpose)
  - 384 vCPU
  - 1536 GiB memory
  - 100000 Megabit network.
  - $25.05984/hour
- p5en.48xlarge (GPUs)
  - 192 vCPU
  - 2 TiB memory
  - 8 H100 GPUs
  - $63.296/hour
- i8g.48xlarge
  - 192 vCPU
  - 1536 GiB memory
  - 12 x 3750GB disks
  - $16.4736/hour



Viewing 1,030 available instances

1 2 3 4 5 6 7 ... 52

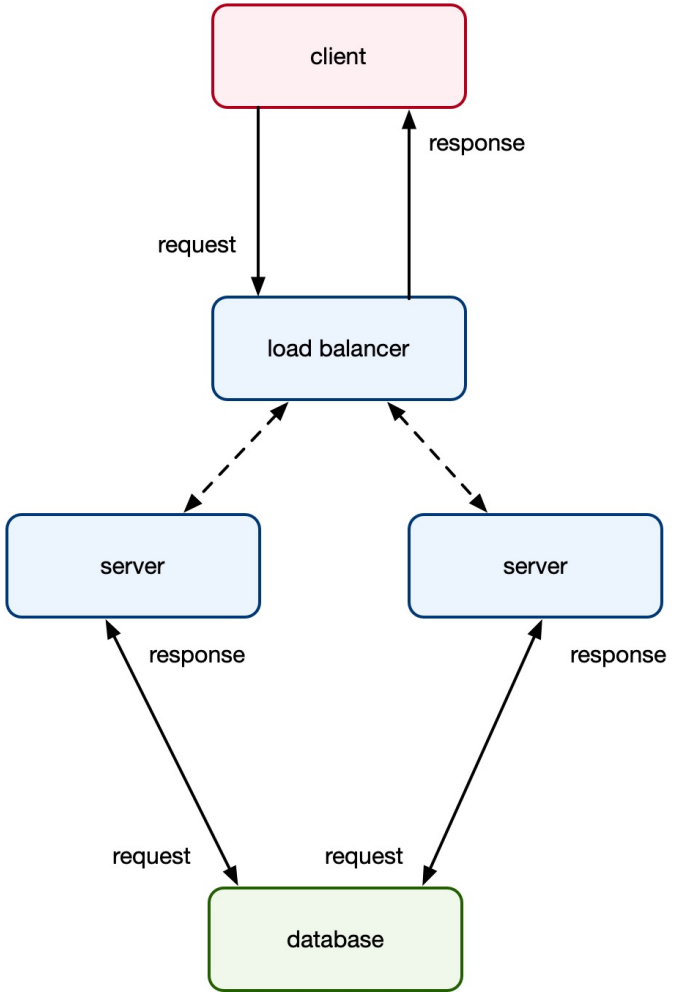
Instance name	On-Demand hourly rate	vCPU	Memory	Storage	Network performance
u7in-24tb.224xlarge	$270.73128	896	24576 GiB	EBS Only	200 Gigabit
u7i-12tb.224xlarge	$125.58182	896	12288 GiB	EBS Only	100 Gigabit
u7i-6tb.112xlarge	$62.79	448	6144 GiB	EBS Only	100 Gigabit
u7i-8tb.112xlarge	$83.72	448	8192 GiB	EBS Only	100 Gigabit
u-6tb1.112xlarge	$54.60	448	6144 GiB	EBS Only	100 Gigabit
m8id.96xlarge	$25.05984	384	1536 GiB	6 x 3800 NVMe SSD	100000 Megabit
r8id.96xlarge	$31.93344	384	3072 GiB	6 x 3800 NVMe SSD	100000 Megabit

Problems with Vertical Scaling

- Cost grows fast.
  - $25/hour is $18,262/month
- Non-cloud solutions have overhead.
  - You have to send someone to the data center to fix the machine when it breaks.
- “Scaling up” can mean “putting your eggs in one basket”
  - Failover?
  - Redundancy?

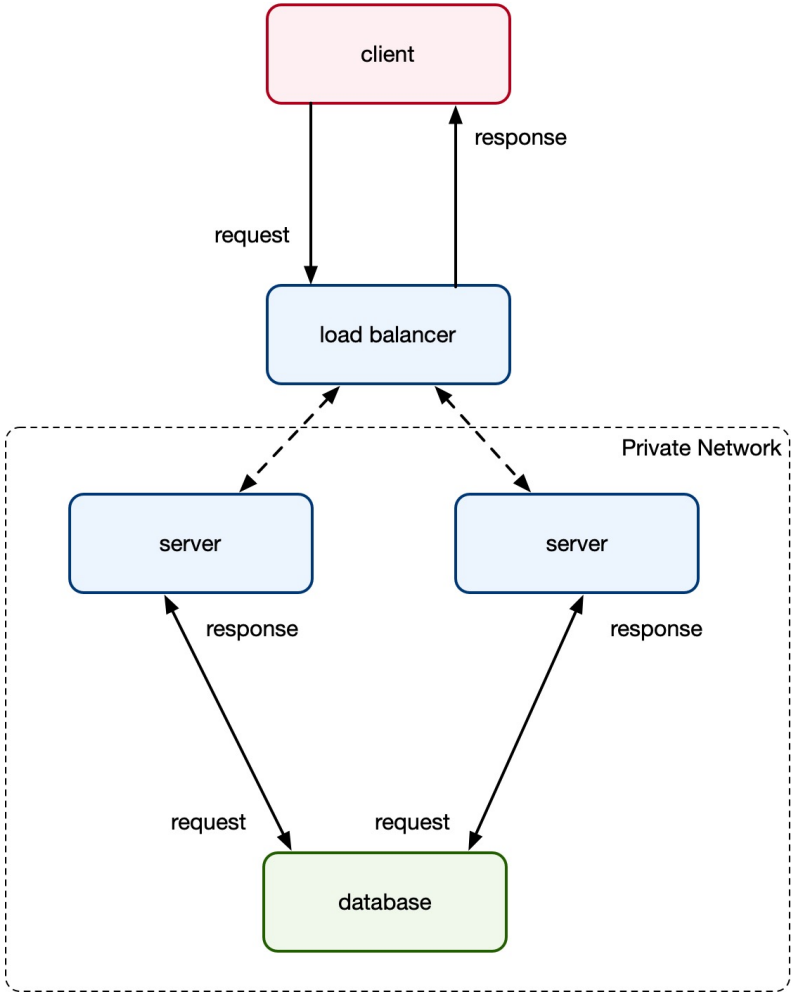
Horizontal Scaling

- Instead of using a single bigger machine, use a pool of several smaller machines.
- This obviously solves the resource problem.
  - Take the resource numbers for a single machine, multiply by size of pool, and you have a good guess at how much you have of each resource now.
- Costs don’t (have to) grow as fast as with vertical scaling.
- What are some challenges?



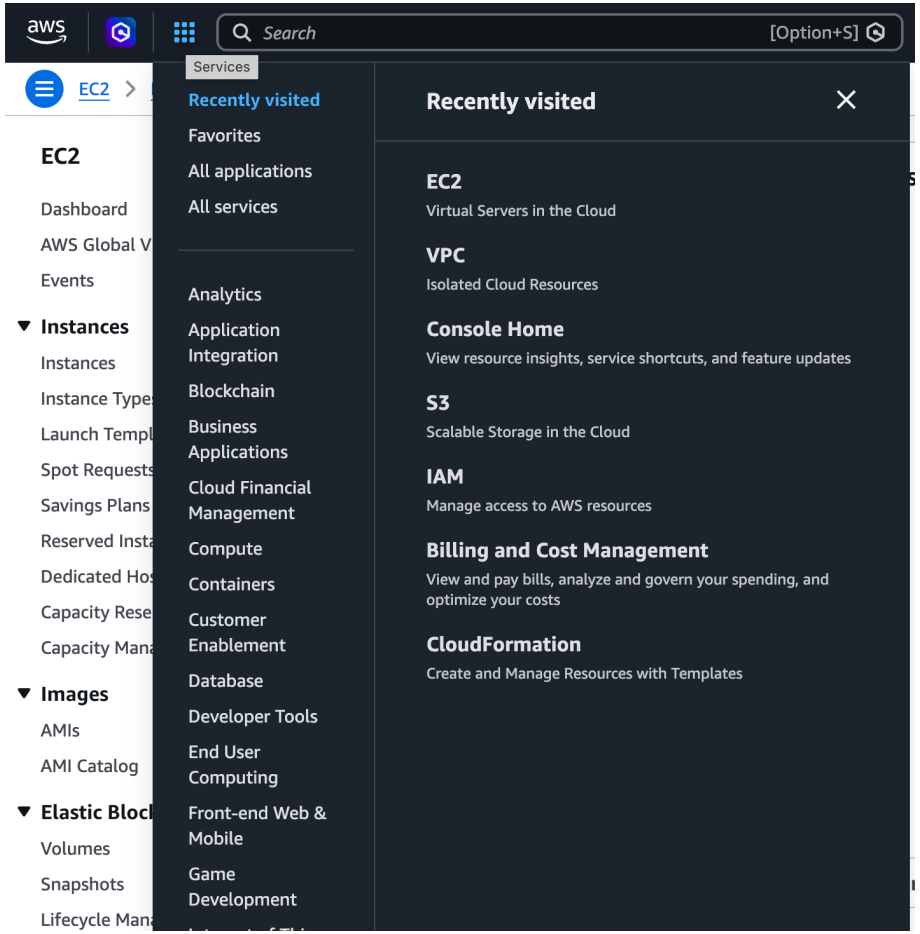
Horizontal Scaling: Private Networks

- Amazon ***Virtual Private Cloud*** lets us define virtual networks that are logically isolated from each other.



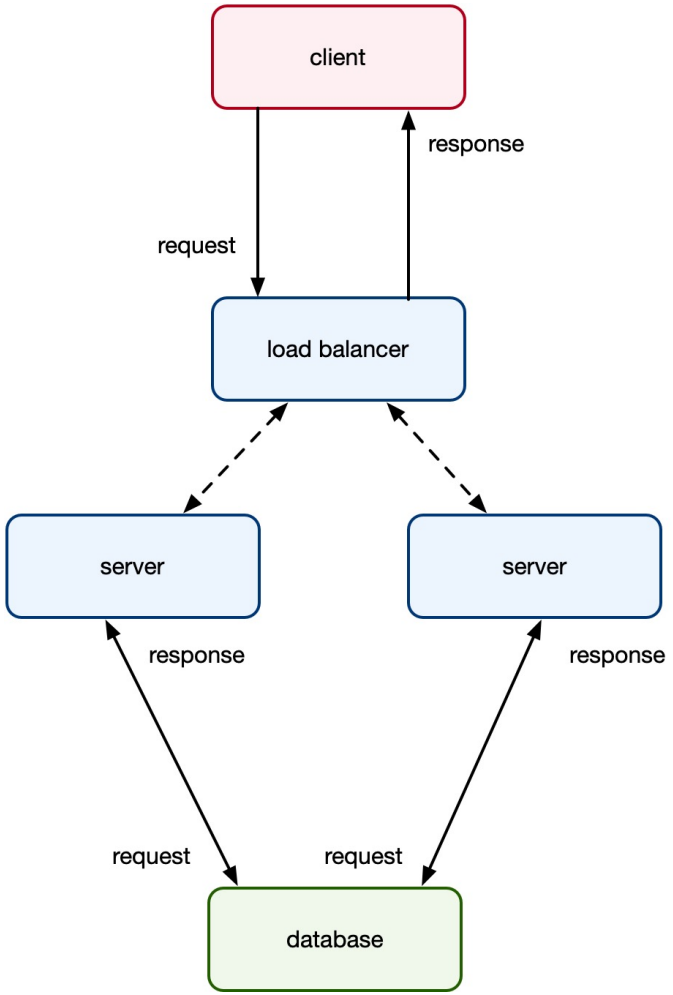
AWS VPC

- Allows to define VPCs and subnets.
- We'll talk about setting up networks on AWS after spring break.



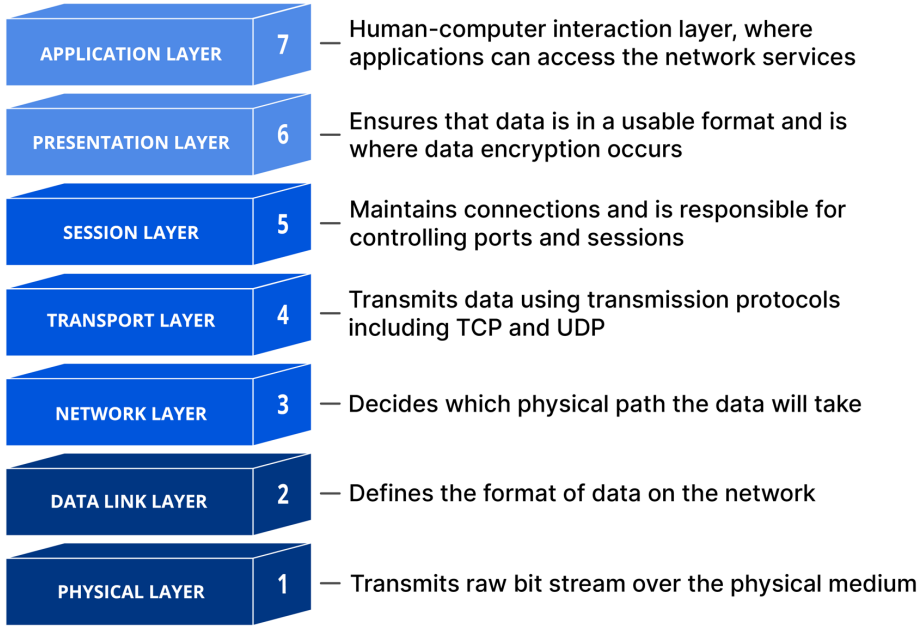
Load Balancing

- A ***load balancer*** is responsible for routing traffic to a particular server.
- Load balancers can forward requests based on several criteria:
- ALB on AWS



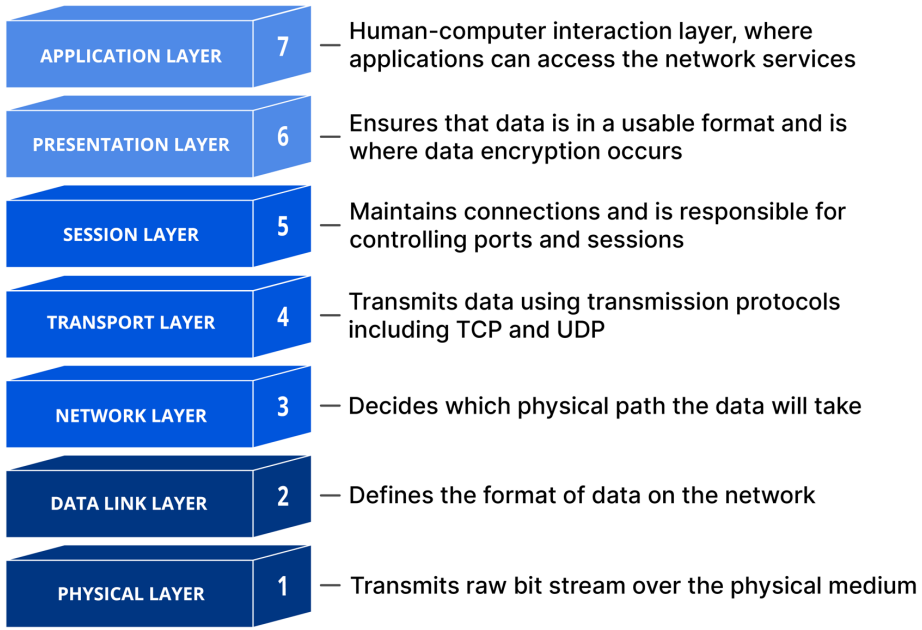
Review: The OSI Model

- Recall that the ***OSI model*** consists of a set of layers that define a communication system (network).
- Two important layers (for us) are the transport layer (layer 4) and the application layer (layer 7).
- TCP is a layer 4 protocol.



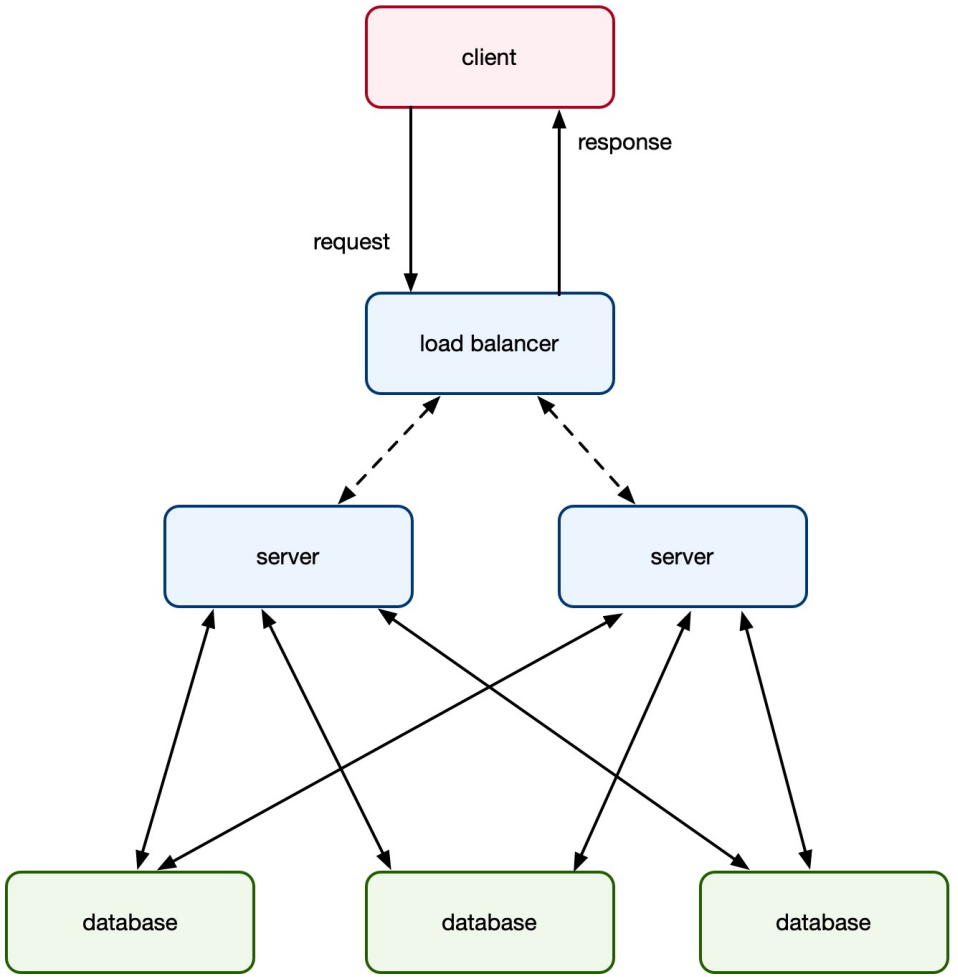
Load Balancers and the OSI Model

- A load balancer has to look at an incoming request and forward it.
- We can distinguish load balancers based on what information they are allowed to use.
- Two main categories
  - L4. Route requests solely based on packet-level information.
  - L7. Route requests based on things like request URL.



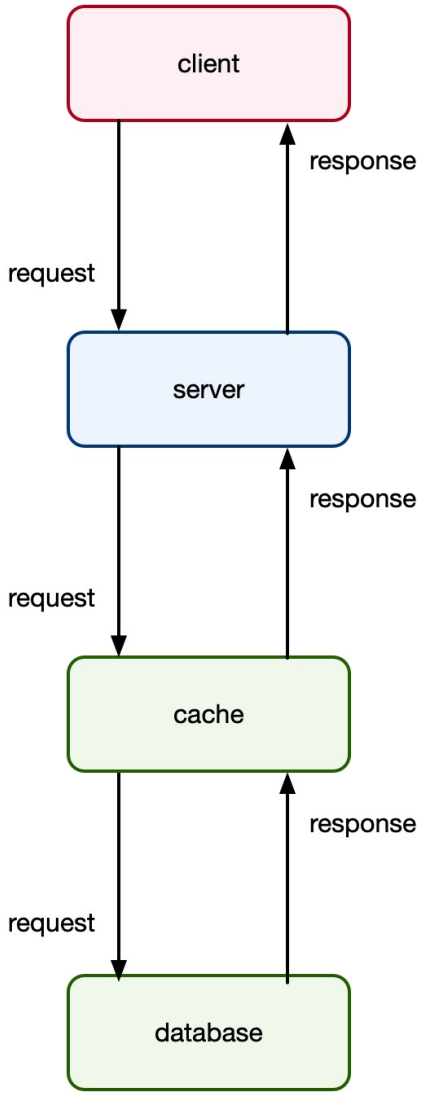
Database Replication

- ***Database replication*** keeps multiple synchronized copies of the same database on different servers.
- In a ***primary–replica model***, one node handles writes and propagates changes (via a log) to replicas.
- Replication improves ***availability*** and read ***scalability***, with tradeoffs in ***consistency***.



Caches

- Database queries can be expensive.
- Requests often have a certain amount of ***locality*** in time or space.
  - A user on a social network may navigate to their “feed” repeatedly in a short period of time.
- A ***cache*** is a program that sits in front of a database and stores recent lookup results.



Content Delivery Networks

- A ***CDN*** (Content Delivery Network) is a geographically distributed network of edge servers that cache and deliver content closer to users.
- ***Static assets*** (images, CSS, JavaScript, video) are replicated across edge locations to reduce latency and offload traffic from the origin server.
- Requests are routed to a nearby edge node using DNS or anycast, minimizing round-trip time.
- CDNs improve performance, reduce origin load, and increase resilience against traffic spikes and DDoS attacks.

Stateful vs. Stateless Architecture

- In a ***stateless architecture***, each request contains all information needed to process it; the server does not retain session state between requests.
- In a ***stateful architecture***, the server maintains client-specific state (e.g., sessions, in-memory data), so subsequent requests depend on prior interactions.

Stateful Architectures

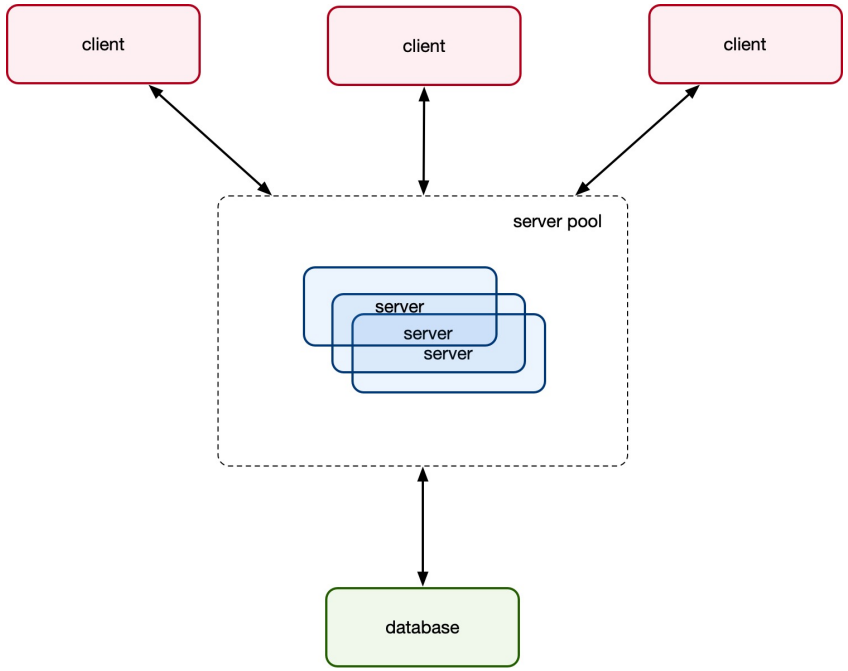
- **Server-side session web apps**: Traditional apps (e.g., PHP, Java Servlets, or Django) store ***user session data*** in server memory or on local disk. Load balancers often require sticky sessions unless session state is externalized.
- **Shopping cart implementations tied to a single app instance**: If cart contents are stored in-process rather than in a shared database or cache, subsequent requests must return to the same server.
- This leads to ***sticky sessions,*** a load balancing strategy in which a client is consistently routed to the same backend server so that server-local session state can be preserved across requests.

Stateless Architectures

- In a **stateless web architecture**, each HTTP request is self-contained: authentication data, parameters, and context (e.g., JWTs) are sent with every request rather than stored in server memory.
- Application servers do not retain per-user session state; any instance behind a load balancer can handle any request interchangeably.
- Persistent state (user data, carts, preferences) is stored in external systems such as databases or distributed caches, not in the application process.
- Stateless designs simplify horizontal scaling and failure recovery: instances can be added, removed, or replaced without disrupting active users.

Stateless Architecture

- All application state is managed through the data tier. The application tier knows nothing.
- This allows us to scale our application tier separately from our data tier.



Data Centers

- We've already seen AWS ***regions*** and ***availability zones***.
- Different AZs are in different data centers.
- Different data centers may be in one AZ, but they're likely to be close to each other.
- It's important to think about regions and AZs in terms of redundancy and fault tolerance.
- Challenges:
  - redirecting traffic, synchronizing data

Message Queues

- A ***queuing system*** decouples producers from consumers by placing tasks or messages into a *durable* buffer that workers process asynchronously.
- Queues smooth traffic spikes by absorbing bursts of requests and allowing worker fleets to scale independently of request arrival rate.
- They improve reliability through persistence and retry semantics: if a worker fails, the message can be re-delivered.
- In cloud environments, managed services such as ***Amazon Simple Queue Service*** or Google Cloud Pub/Sub provide elastic scaling, visibility timeouts, and dead-letter queues for fault isolation.

Logs and Metrics

- **Logging** records discrete events emitted by applications and infrastructure (errors, requests, state transitions), providing detailed forensic data for debugging and auditing.
- **Metrics** capture quantitative time-series measurements (CPU usage, request latency, error rate), enabling aggregation, dashboards, and alerting.
- At the host level, you collect OS and process telemetry.
- At the application level, services emit structured logs and application metrics.
  - Open-source tools like Prometheus facilitate this.
- At the cluster or service level, you move from machine health to system health

Scaling Databases

- **Vertical scaling (scale up)** increases the resources of a single database node
  - more CPU cores,
  - RAM,
  - faster storage (IOPS/throughput).
  - It is simple operationally but bounded by hardware limits and often requires downtime or failover.
- **Horizontal scaling (scale out)** adds more database nodes, distributing load across replicas or shards. This increases capacity and availability but introduces coordination complexity.
  - Horizontal scaling may involve **read replicas** (for read throughput), or
  - **sharding/partitioning** (for write scalability), which require routing logic and careful data modeling.

Horizontal Database Scaling

- **Read scaling via replication**: Additional replica nodes serve read queries while a primary handles writes. This increases read throughput but does not inherently increase write capacity.
- **Sharding (horizontal partitioning)**: Data is split across multiple nodes by key range or hash (e.g., user ID modulo N). Each shard handles a subset of writes and reads, increasing total write capacity.
- Challenges
  - **Routing layer required**: Applications or middleware must determine which shard holds a given record; consistent hashing or directory services are common strategies.
  - **Cross-shard queries are complex**: Joins, transactions, and aggregates across shards require coordination, often reducing performance or forcing application-level handling.
  - **Rebalancing and resharding**: Adding nodes requires redistributing data, which can be operationally expensive and must be managed carefully to avoid hotspots and downtime.

Vertical Database Scaling

- **Single-node resource expansion**: Increase CPU, RAM, and storage performance (e.g., higher IOPS, larger buffer pool) on one database server to handle greater query volume and larger working sets.
- **Memory as a primary lever**: More RAM allows a larger cache/buffer pool, reducing disk I/O and dramatically improving read and index performance.
- **Storage performance tuning**: Upgrading to faster disks (NVMe, provisioned IOPS volumes) improves transaction log writes and random read latency.
- **No data distribution complexity**: All tables, indexes, and transactions remain on one node, so ACID guarantees and query planning remain straightforward.
- **Hard upper bound**: Capacity is limited by the largest available hardware configuration; beyond that, horizontal techniques (replication, sharding) become necessary.

Example Architectures

Example: Learning Management System (LMS)

- We’ve all used Brightspace.
- What would we need to do to build it?

Example: LMS High-Level Architecture

- **Users**: students, instructors, administrators access the LMS via browser or mobile app.
- **Frontend**: web application (HTML/CSS/JS) renders courses, assignments, grades.
- **Application layer**: backend services implement authentication, course logic, grading workflows.
- **Data layer**: relational database stores users, enrollments, submissions, grades.
- **File/object storage**: stores uploaded assignments, videos, course materials.
- **Optional integrations**: email service, video conferencing, plagiarism detection.

Example: LMS Frontend and Delivery Layer

- CDN caches static assets (JS bundles, images, CSS) to reduce latency globally.
- Reverse proxy/load balancer distributes traffic across multiple app servers.
- Stateless application servers process requests; authentication handled via cookies.
- WebSocket or long-polling channels may support real-time notifications (e.g., announcements).

Example: LMS Application Layer

- Authentication & authorization service enforces roles (student vs instructor).
- Course management service handles enrollments, rosters, content modules.
- Assignment service manages submissions, deadlines, and grading states.
- Background workers process asynchronous tasks (email notifications, file processing, grade exports).
- Queuing system decouples long-running jobs from user-facing requests.

Example: LMS Data and Storage Layer

- Primary relational database (e.g., PostgreSQL) stores structured data with ACID guarantees.
- Read replicas may support heavy reporting workloads (analytics dashboards, grade summaries).
- Object storage (e.g., Amazon Simple Storage Service) holds large files such as PDFs and videos.
- Backup and snapshot systems ensure durability and disaster recovery.

Example: LMS Scaling and Reliability

- Horizontal scaling of application servers behind a load balancer for peak usage (e.g., exam deadlines).
- Database vertical scaling for transactional integrity; optional horizontal scaling via replication.
- Monitoring and logging systems track latency, error rates, and resource utilization.
- Automated failover and health checks maintain availability during node failures.

Example: An LLM Chat Service

- How is something like ChatGPT implemented *as a website?*
- What do we need to build?
- How is it different from an LMS?

Example: LLM Chat High-Level Architecture

- Users access the chat service via web or mobile client.
- Frontend communicates with a backend API over HTTPS.
- API layer handles authentication, rate limiting, and request validation.
- Inference layer runs the large language model (LLM).
- Data layer stores conversations, user accounts, and usage metadata.

Example: LLM Chat Frontend and Edge Layer

- CDN caches static assets (JS, CSS) for global performance.
- Load balancer routes API calls to stateless backend instances.
- Each chat request includes authentication credentials (e.g., JWT).
- Streaming responses (server-sent events or WebSockets) deliver tokens incrementally to the client.

Example: LLM Chat API and Orchestration Layer

- Chat API receives prompt + conversation history.
- Request validation enforces token limits and safety filters.
- Orchestrator formats input into model-compatible prompt structure.
- Rate limiting and quota enforcement protect GPU resources.
- Requests are queued if inference capacity is saturated.

Example: LLM Chat Inference Layer

- LLM runs on GPU-backed instances.
- Model weights are loaded into GPU memory; KV-cache stores intermediate attention state per request.
- Batch scheduling groups multiple user requests to maximize GPU utilization.
- Horizontal scaling adds more inference nodes; a router distributes requests across them.

Example: LLM Chat Data and Persistence

- Relational database (e.g., PostgreSQL) stores user accounts, billing info, metadata.
- Conversation history may be stored in a database or document store.
- Object storage holds model artifacts and logs.
- Caching layer (e.g., Redis) supports session tokens, rate counters, and short-term conversation context.

Example: LLM Chat Scaling and Reliability

- Stateless API servers scale horizontally behind a load balancer.
- GPU inference nodes scale independently from the API layer.
- Queuing system smooths traffic spikes and prevents overload.
- Health checks and autoscaling policies respond to demand (e.g., peak classroom usage).

Example: LLM Chat Observability and Safety

- Metrics track latency (time to first token, tokens/sec), GPU utilization, and error rates.
- Structured logs capture prompts (with privacy controls), moderation results, and failures.
- Safety filters run pre- and post-inference to enforce content policies.
- Alerting systems trigger on abnormal latency, cost spikes, or model instability.

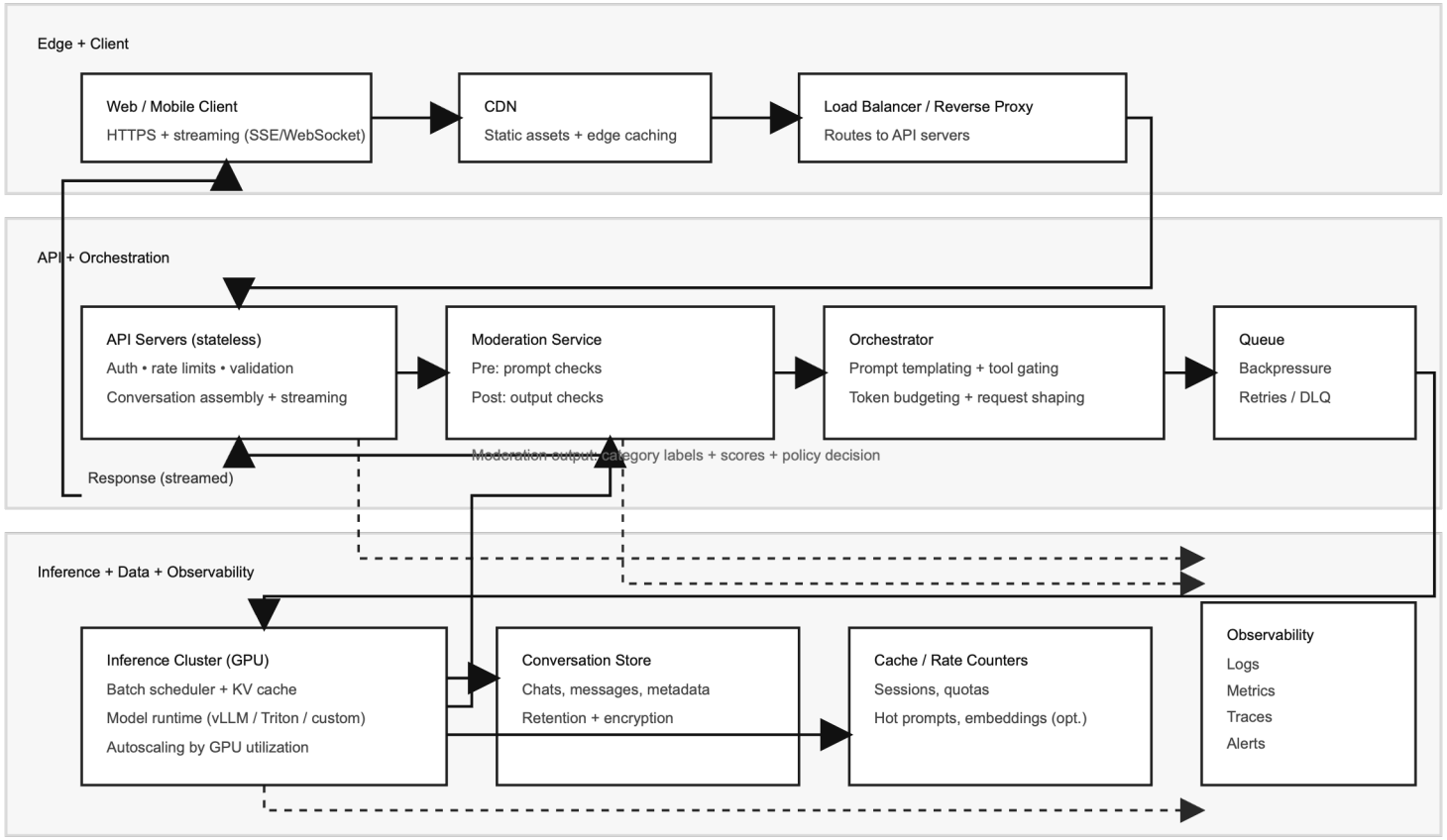
Example: LLM Chat Safety Filters and Moderation Pipeline

- **Pre-inference moderation**: Incoming prompts are scanned before reaching the model. Classifiers detect categories such as violence, self-harm, sexual content, hate speech, or prompt injection attempts. High-risk inputs may be blocked, rewritten, or escalated.
- **Prompt sanitization and policy enforcement**: The system may strip disallowed instructions, enforce system-level guardrails, or constrain tool access before constructing the final model prompt.
- **Post-inference moderation**: The generated output is re-evaluated by a separate moderation model. Even if the input was safe, the output may be filtered, truncated, or replaced if it violates policy.
- **Category labeling and confidence scores**: Moderation systems typically return structured results (e.g., category flags plus probability scores), enabling threshold-based decisions rather than binary blocking.

Example: LLM Chat Safety Filters and Moderation Pipeline

- **Streaming-time intervention**: For token-streamed responses, moderation may run incrementally; generation can be halted mid-stream if unsafe content is detected.
- **Audit logging of moderation results**: Systems log category labels, scores, policy decisions (allow/block/transform), and model version. This supports compliance, appeals workflows, and tuning of thresholds.
- **Human-in-the-loop escalation**: Edge cases or high-severity categories may trigger review queues for human moderators, especially in regulated or educational contexts.
- **Defense-in-depth design**: Safety controls exist at multiple layers—API validation, system prompts, model fine-tuning, output filters, rate limits—so failure of one mechanism does not immediately result in policy violation.

Example: LLM Chat



Is the SVG above a good description of how you are implemented?

At a high level, yes — it's a reasonable **conceptual architecture** for a modern LLM chat service. It captures the right abstractions:

- Edge layer (client, CDN, load balancer)
- Stateless API layer
- Moderation before and after generation
- Queueing and backpressure
- GPU-backed inference cluster
- Persistence + caching
- Observability as a cross-cutting concern

That said, it is still a **simplified teaching diagram**, not a literal blueprint of my internal deployment. Real systems typically add:

- Multiple model tiers (small routing model → large generation model)
- Dynamic model selection and routing
- Separate embedding services
- Multi-region replication and traffic steering