Process

1. Gather Functional & Nonfunctional Requirements
    - **Functional**: what functionality (who will use it, what features do we need e.g. accounts, payment system, analytics)
    - **Nonfunctional**: availability vs consistency (e.g. social media vs banks), latency (e.g. batch vs streaming), performance, throughput
        - How many users (e.g. MAU), how many daily actions (e.g. checkouts), how much daily storage generated (e.g. tweets made)
    - **Calculations**: byte (one text char) → thousand bytes = 1 KB → million bytes = 1MB → billion bytes = 1GB → trillion bytes = 1TB
        - avg text paragraph = 10KB, avg image = 5MB, avg SD movie = 2GB, avg 4K movie = 20GB, 50% of users leave if 2+ sec page load latency
2. Design Public & Private API Endpoints
    - **API**: Type (REST, RPC, GraphQL), API Name (endpoint name e.g. /reserve, /payment, /cancel), API Params, Return, Notes
    - **Parking Lot Ex**: Endpoint: /free_spots Param: garage_id, vehicle_type, time Return: spot_id Note: smaller vehicle can fit in larger spots
3. Design Database Model
    - RDBMS When: relational (1-1, 1-Many, Many-Many), complex joins, high consistency transactions
        - Schema: Table (per object e.g. users, garage, spots, reservations), Col Names, Dtypes (pk, id, fk, int, bool, time, enum, decimal, varchar)
    - NoSQL When: un/semi-structured data, big data that needs to scale, high availability
        - Key-Val: (design key-vals e.g. key=celeb profile URL, val=generated HTML) Doc: (design docs fields like JSON) Wide Col: (design fields)
4. Design & Scale a Sytem
    - **Initial System**: client → DNS → web server → read & write APIs → DB → object storage
    - **Scale System**: load balancers, CDN, caches, sharding, read replicas, microservices, queue → asynchronous, big data (Kafka, Hadoop, Spark)
    - **Build Microservices**: (e.g. ML recommender system, checkout, promotions, notifications)

Topics

- Tradeoffs
    - CAP: **Consistency** (read gets most recent write or error) **Availability** (read can get old write) **Partition Tolerance** (system survives outages)
    - Consistency: **Weak** (reads miss a write, e.g. call) **Eventual** (reads get write async, e.g. email) **Strong** (reads get write sync, e.g. transactions)
    - Availability Failover: **Active-Passive** (try active, busy use passive, can get old data) **Active-Active** (multiple active servers distribute requests)
    - Availability Replication: **Master-Slave** (create read replicas of master) **Master-Master** (masters handle read & write, slow writes because sync)
    - Latency vs Throughput: Aim for maximal **throughput** (num of action per unit of time) with acceptable **latency** (time to perform action)
- Load Balancers
    - **LB**: distribute incoming client requests to computing resources
    - **(+)** make easy to horizontally scale, prevent overloading resources, LB has public IP allowing compute resources to have private IP
    - **Scale**: have multiple LBs using Failover technique **Reduce Latency**: check a cache for read requests before querying DB
    - **Routing**: Round Robin, Least Loaded, Consistent Hashing (won't have to clear caches when adding/removing workers)
- Caching & CDN
    - In memory caching reduces read latency and load on DBs/microservices (e.g. cache a query result, HTML page, user session)
    - **Eviction**: use LRU or TTL by default **Types**: Redis (OS, nosql key-val, persists to DB in case of crash), Memcached
    - Patterns: **Cache-Aside**: check cache, if miss grab from DB and add to cache /w TTL **Write-Through**: write to cache if miss, then sync write to DB
    - **CDN**: globally distributed network of proxy servers **(+)** users get content from nearby servers **(-)** content might be stale if updated before TTL
    - **Pull CDN**: cache content first time user requests, purge when TTL expires (reduces CDN storage) **Push CDN**: recache content whenever updated
- Microservices
    - **Monolith**: singular, large computing network, one code base, all services coupled, communicate via function calls
    - (+) no network calls → services communicate faster **(-)** single point of failure, can't scale individual components, slower deployment
    - **Microservices**: independently deployable modular services in application layer (e.g. cart, payment, analytics, ML), communicate via RPC/REST
    - **(+)** can scale individual services, continuous deployment **(-)** network calls → slower, higher complexity (separate code bases, tech stacks)
- Databases

- RDBMS: store tabular data using a data schema and ACID rules into relational tables that reference one another using foreign keys
  - ACID: **Atomic** (all or nothing) **Consistent** (strong) **Isolated** (concurrent transactions same as sequential) **Durable** (persist system failures)
  - Scale: **Replication** (Master-Slave, Master-Master) **Sharding** (partition into smaller DBs on separate servers, e.g. split on last name, country)
  - **Federation** (split DB by function into smaller DBs on separate servers, joins require server link, e.g. split into users, products, purchases)
  - **Denormalization** (store redundant copies of columns in tables to min joins) **SQL Tuning** (e.g. split BLOBs into own table using federation)
  - **(+)** strong consistency, consistent schema, complex joins, transactions **(-)** scaling makes joins harder, changing schema is expensive
- NoSQL: store unstructured data with a dynamic schema, give up ACID for high scalability and high availability but eventual consistency
  - BASE: **Basically Available** (high availability) **Soft State** (data can be temporarily inconsistent) **Eventual Consistency** (reads get write async)
  - Types: **Key-Val**: O(1) read/write, e.g. Redis in-mem caching **Document**: key-val, val is semi-struct doc (JSON) e.g. MongoDB, ElasticSearch
  - **Column**: flexible schema, rows → super col family → col family → col e.g. Cassandra, BigTable **Graph**: complex networks e.g. Neo4j
  - **(+)** easy to horizontally scale since non-ACID, unstructured data → flexible schema, big data **(-)** eventual consistency, joins are very difficult

- Communication
  - Network Stack: **Application** (HTTP, XMPP, SSL, TLS, DNS) → **Transport** (TCP, UDP) → **Network** (IPv4, IPv6) → **Link** (Ethernet, WiFi)
  - Request-Response APIs
    - **REST**: external endpoints, return resources (e.g. user), CRUD using GET, POST, PUT, DELETE, PATCH, (-) resources → return large payloads
    - **RPC**: internal endpoints, perform actions, use GET & POST, (+) higher performance since lighter payloads, (-) endpoints not as standardized
    - **GraphQL**: send server custom data structure /w fields you want, use for complex data, (+) customizeable, no API versions, light payloads
  - Event-Driven APIs - use when you have to wait on server to complete process before returning result to client
    - **WebHooks**: client provides callback URL and subscribes to API events, e.g. webhook alerts when email bounces
    - **WebSockets**: long lived low-latency bidirectional client-server connection, (+) no need to resend HTTP requests, e.g. gaming, chat
    - **HTTP Streaming**: client sends one request to server, server reponds with data stream, e.g. Twitter gathers new tweets when user connects
  - Transport Protocols
    - **TCP**: connection over IP, packets guaranteed to arrive uncorrupted and in order otherwise resent, e.g. download an image or document
    - **UDP**: lower-latency but no packet arrival guarantee, use for streaming real-time data, e.g. gaming, video chat
- Asynchronism
  - **Queues**: in memory task/message queues enable async handling of tasks, queues recieve, hold, & deliver tasks to compute workers
  - **Types**: Kafka, Redis Queue, Celery **Fault Tolerance**: persist in progress tasks to DB **Back Pressure**: if queue fills up, return error to client
  - **Concurrency**: one core juggling multiple threads over time, one thread gets run at a time until a pause, then switched out for a hanging thread
  - **Locking**: Lock a thread to force synchronous processing **Parallelism**: multiple cores handling multiple threads at the same time
  - **Async Ex**: users upload videos, videos sent to task queue, users can close browser, workers async process video, email back result video
- Data Processing
  - **Batch**: process data on a schedule (e.g. process EOD server logs) **Stream**: process data in real time (e.g. real-time analytics, bank transactions)
  - **MapReduce**: split-apply-combine big data sets in parallel on a cluster, 1. map: sort chunks across nodes, 2. reduce: perform op on each node
  - Big Data Processing /w Hadoop or Spark
    - **Hadoop**: read/write files in duplicated chunks to HDFS cluster, use MapReduce for batch processing, use Yarn to coordinate cluster nodes
    - **Spark**: MapReduce but in memory, used for real-time processing, (+) much faster than Hadoop, less code than Hadoop
  - Data Streaming /w Kafka (AWS Kinesis similar)
    - **Kafka**: act as task queue, transport event driven data streams from data producers to data consumers in a distributed low latency format
    - **Data Streams Ex**: website events (clicks), transaction data, Uber sends real-time geolocation data to ML services (ETA, surge pricing)