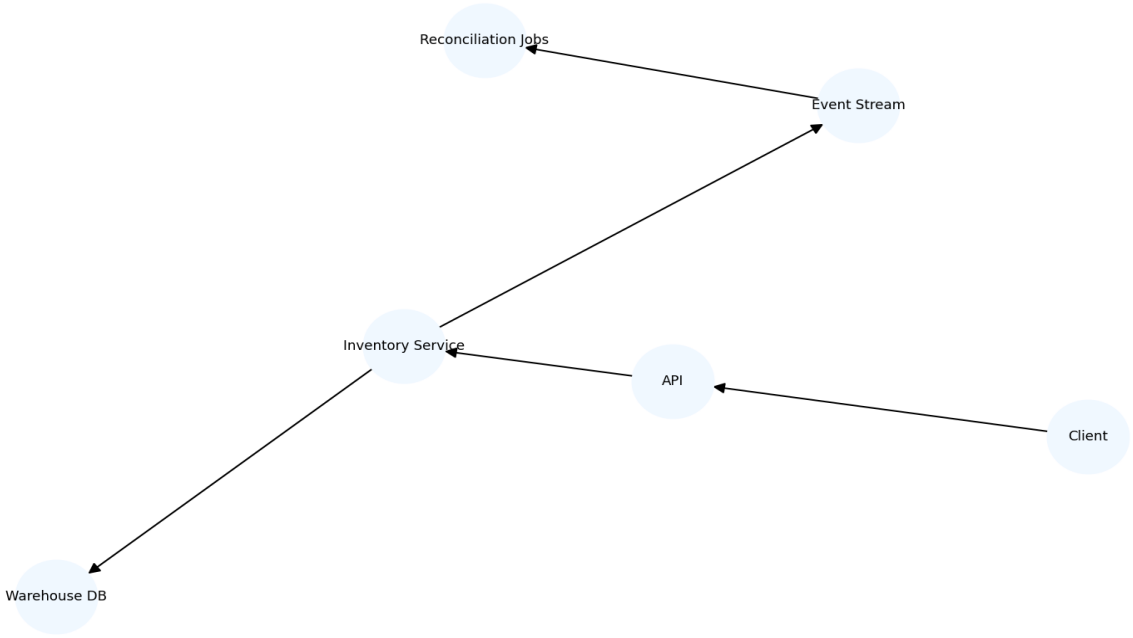
Amazon & Facebook System Design — Answers and Diagrams

Inventory Management System (Amazon)

Assumptions: Millions of products, multiple warehouses, concurrent orders, per-SKU stock, eventual consistency for most operations but strong for reservations.
APIs: GET /inventory/{sku}, POST /reserve {sku,qty,orderId}, POST /release, POST /adjust
Design: Central Inventory Service with per-warehouse partitions. Use local stock caches, global index for availability. Reservations use conditional writes with TTL. Async reconciliation jobs sync across warehouses.
Scaling: Shard by warehouse and SKU hash. Use read-replicas and cache hot SKUs. Use CQRS: fast reads from cache, writes go to authoritative store and event stream.
Failure Modes: Network partitions: fall back to local warehouse mode; double-reserve prevention via idempotency keys and reservations TTL.
Security: IAM for services, audit logs for adjustments.
Trade-offs: Strong consistency for checkout vs higher latency; choose reservations to localize locks.

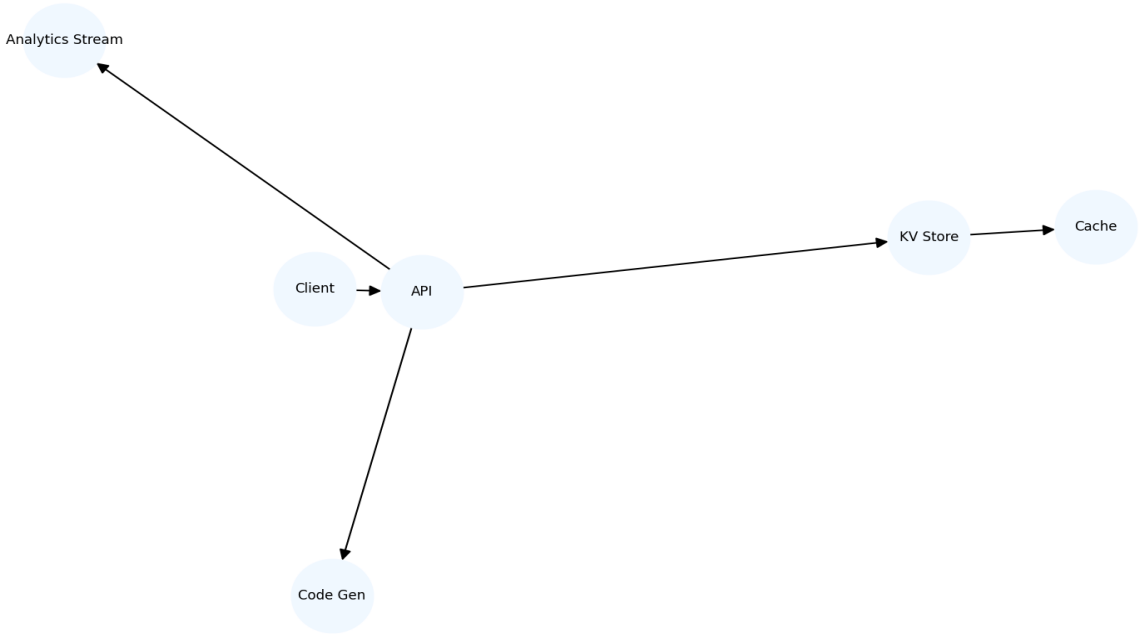
Start with requirements, show per-warehouse partitioning, reservation flow (reserve -> commit -> release), and reconciliation.



URL Shortener (Bitly-like)

Assumptions: High read:write ratio, low latency redirect, custom aliases allowed, analytics optional.
APIs: POST /shorten, GET /{code}, GET /analytics/{code}
Design: API layer -> Code generation service -> Primary KV (Dynamo/Aurora) with Redis cache. Edge redirect service uses CDN/edge cache for hot codes. Event stream for analytics.
Scaling: Shard by code prefix; cache hot codes in memory; use bloom filters to route misses efficiently.
Failure Modes: Cache misses handled via DB fallback; custom alias collisions handled at creation time.
Security: Abuse detection, rate limits, URL validation.
Trade-offs: Immediate consistency vs eventual for analytics; centralization for custom alias checks.

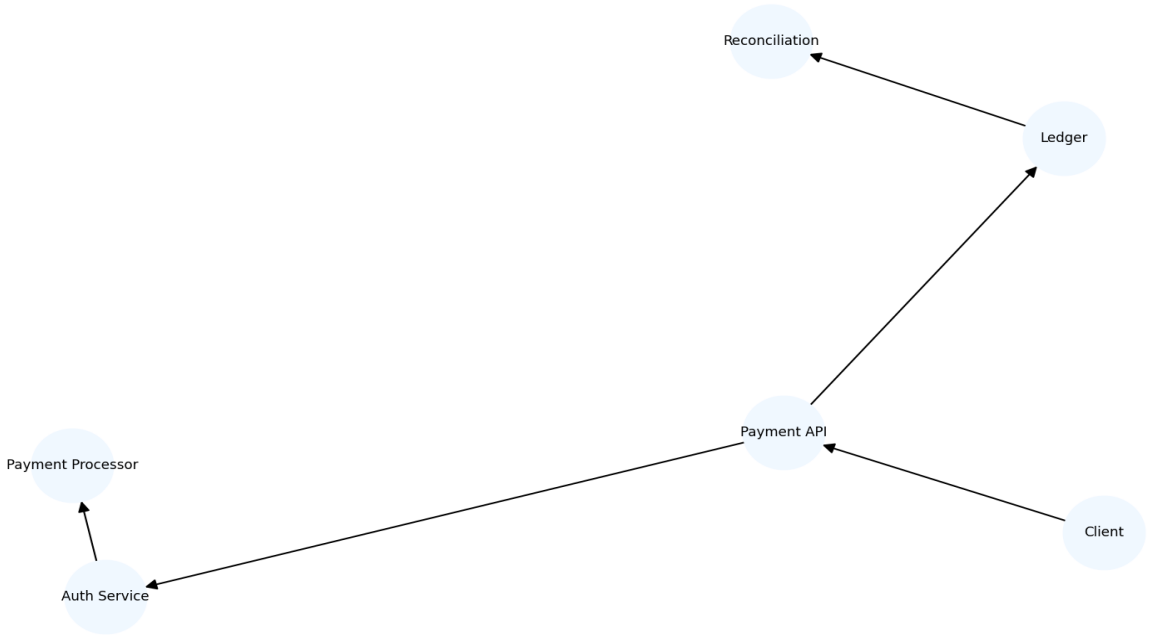
Describe code gen, persistence, caching, edge redirect, and analytics pipeline.



Online Payment System (Amazon Pay)

Assumptions: High reliability, PCI compliance, auth/capture flow, fraud detection.
APIs: POST /pay/authorize, POST /pay/capture, POST /refund, GET /status
Design: Payment gateway façade handles tokenized cards, routes to payment processors. Separate services for auth, capture, ledger, reconciliation, and fraud. Use outbox pattern for reliable events.
Scaling: Stateless API layer, stateful ledger DB with partitioning by merchant or account, queues for async reconciliation.
Failure Modes: Processor failures: retry with backoff, switch processors; ensure idempotency and eventual reconciliation.
Security: PCI-DSS, tokenization, encrypted storage, strict audit trails.
Trade-offs: Sync auth for user experience vs async settlement for backend reconciliation.

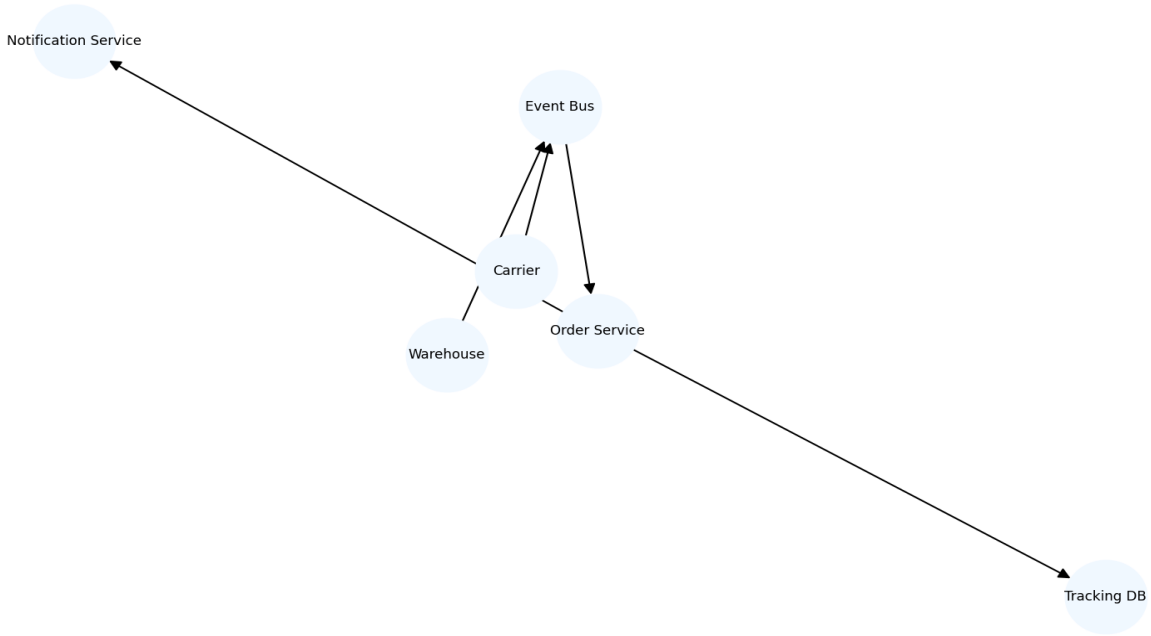
Explain auth/capture flow, ledger writes, reconciliation, and fraud checks.



Order Tracking System

Assumptions: Millions of orders/day, real-time status updates, customer-facing tracking pages.
APIs: GET /order/{id}, POST /order/{id}/status, webhooks for carriers
Design: Order Service stores status timeline; events from fulfillment/warehouse/carriers flow into an event bus; read-optimized tracking DB (materialized views) serves queries; notification service pushes updates.
Scaling: Partition orders by orderId ranges; use CQRS with event sourcing to enable replay and consistency; cache recent orders.
Failure Modes: Late carrier updates handled via reconciliation; eventual consistency for tracking vs immediate for critical updates.
Security: Auth, signed webhooks between partners.
Trade-offs: Real-time accuracy vs cost of constant polling of carriers.

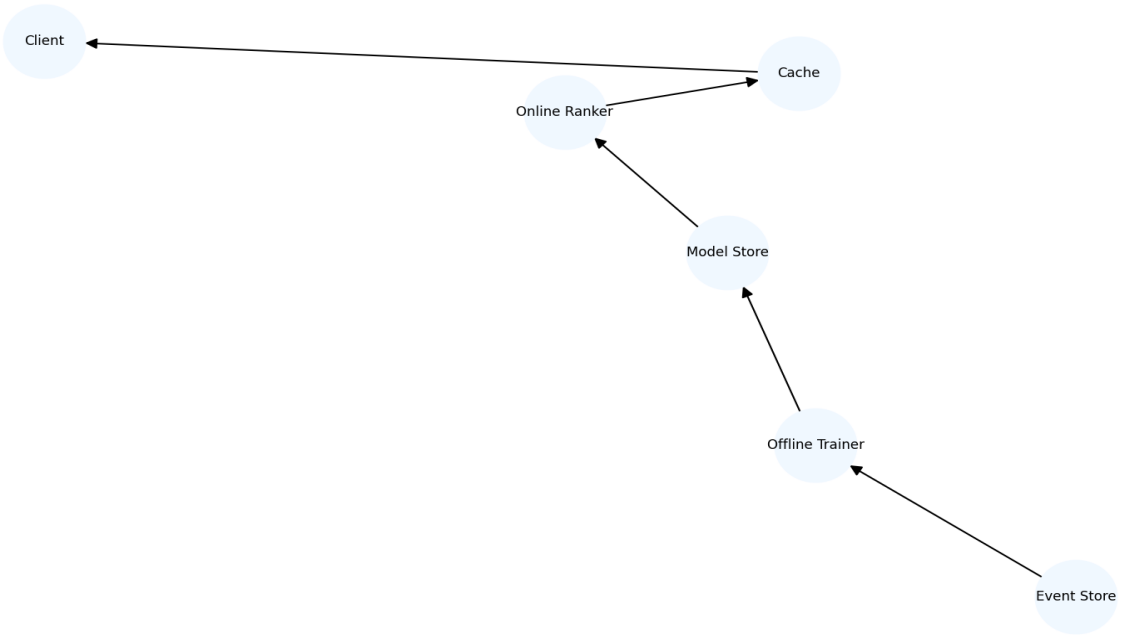
Show event-driven flow: sources -> event bus -> materialized views -> API.



Recommendation System (Amazon)

Assumptions: Large catalog, collaborative filtering + item-based recommendations, offline model training, online feature serving.
APIs: GET /recommendations?userId, batch training jobs
Design: Offline batch pipeline to compute embeddings and candidate lists; online ranking service uses user features and recent activity; feature store for serving features; A/B testing pipeline.
Scaling: Precompute per-user or per-segment recommendations, use caching and personalization shards. Use approximate nearest neighbors for similarity lookup.
Failure Modes: Cold-start handled via popularity baselines; stale models gradually roll out with canary testing.
Security: Privacy controls and data retention.
Trade-offs: Real-time personalization vs precomputed recommendations (latency vs freshness).

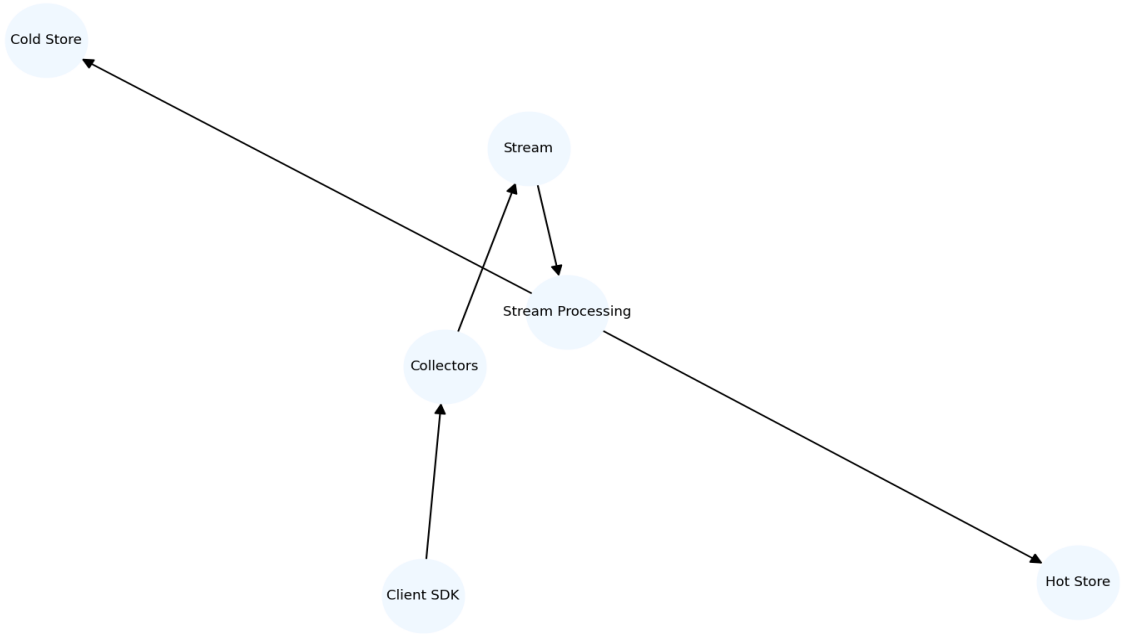
Outline offline training -> candidate generation -> online ranking -> feature serving.



Real-time Analytics Pipeline

Assumptions: Clickstream at high throughput, need near-real-time metrics, batch backfills.
APIs: Ingest endpoints, query endpoints (dashboards)
Design: Client SDK -> regional collectors -> partitioned stream (Kafka/Kinesis) -> stream processing (Flink) -> hot store (OLAP) and cold store (data lake). Materialized views updated for dashboards.
Scaling: Partition by customerId or session, autoscale consumers, use compacted topics for stateful operations.
Failure Modes: Backpressure: buffer & backoff; consumer lag alerting; replay via retained topics.
Security: Access control, PII scrubbing at ingestion.
Trade-offs: Window size and latency vs compute cost.

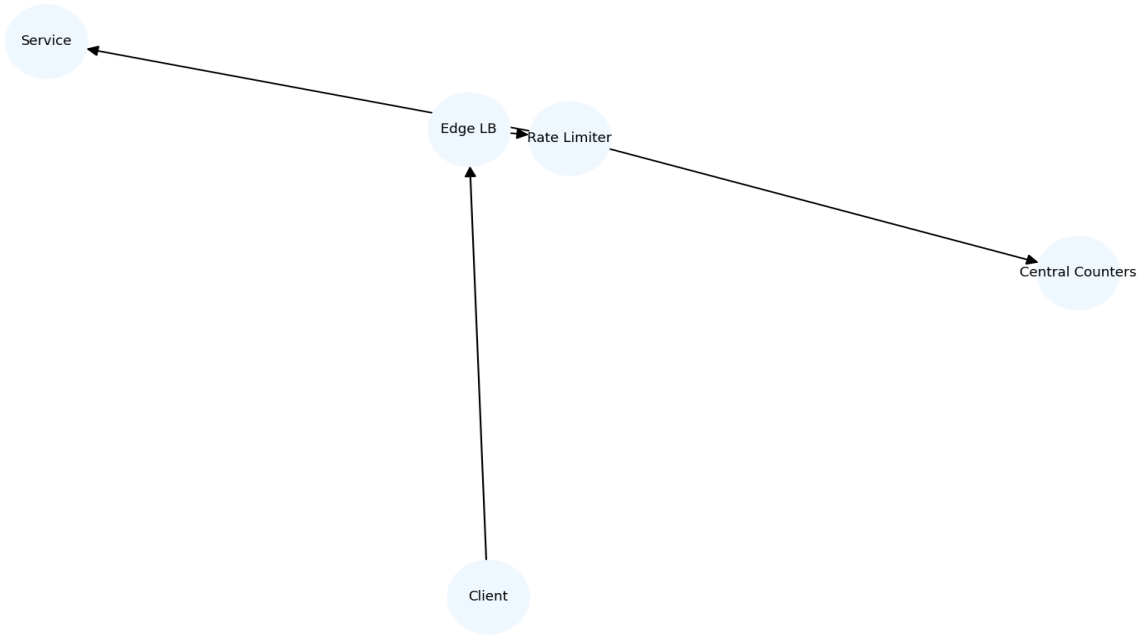
Walk ingest -> stream -> processing -> serving for dashboards.



API Rate Limiter (high QPS)

Assumptions: Millions RPS, per-user and global limits, distributed services
APIs: Enforced within gateway: token bucket responses
Design: Edge rate limiter (local per-node token buckets) backed by a central counter for global rules. Use leaky bucket/token bucket with approximate enforcement via local caches and periodic syncs. For strict limits, use centralized Redis with sharding.
Scaling: Distribute limits by user key, use Bloom filters for unknown keys, local caches for hot clients.
Failure Modes: Network partitions: fail-open or fail-closed policy depending on safety; use conservative default limits.
Security: Prevent abuse with anomaly detection.
Trade-offs: Accuracy vs scalability: local approximations improve performance but can slightly exceed limits.

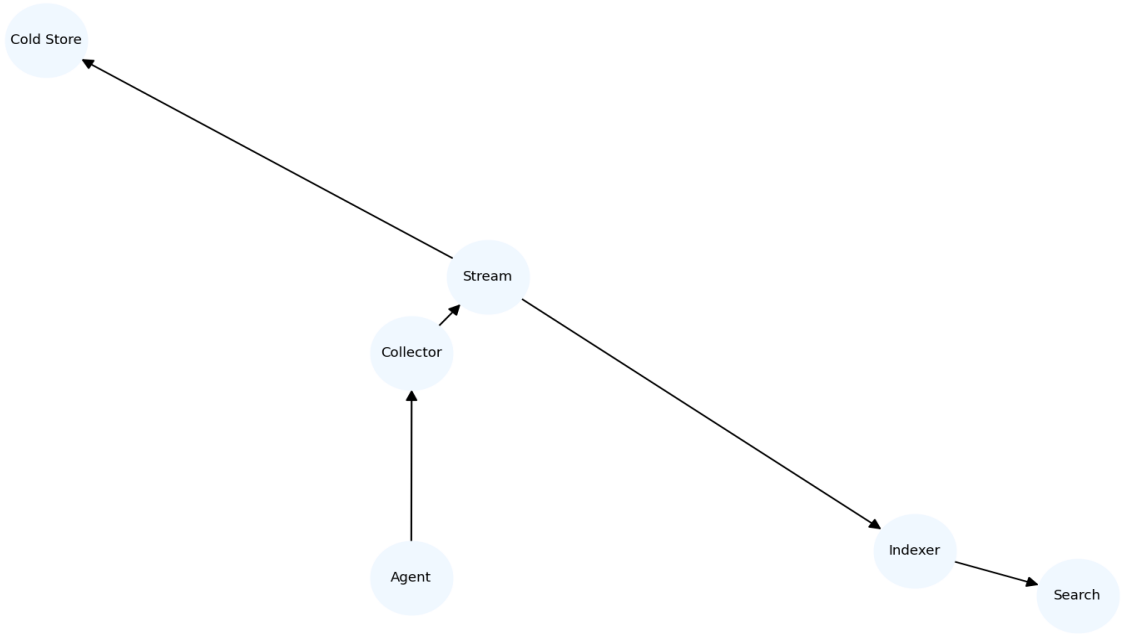
Explain local token buckets, sync, and handling strict vs approximate limits.



Scalable Logging System

Assumptions: High log volume, need for retention, search and alerting
APIs: Ingest logs, query logs, alerting hooks
Design: Agent -> collector -> partitioned stream -> storage (hot index for recent logs, cold blob store for long-term). Indexing and search layer (Elasticsearch/ClickHouse), retention policies and rollups for cost control.
Scaling: Sharded indexes, ILM (index lifecycle management), tiered storage
Failure Modes: Backpressure, sampling to reduce volume, durable buffers on collectors
Security: Sensitive data redaction, access control, audit logs
Trade-offs: Full fidelity vs sampled logs for cost.

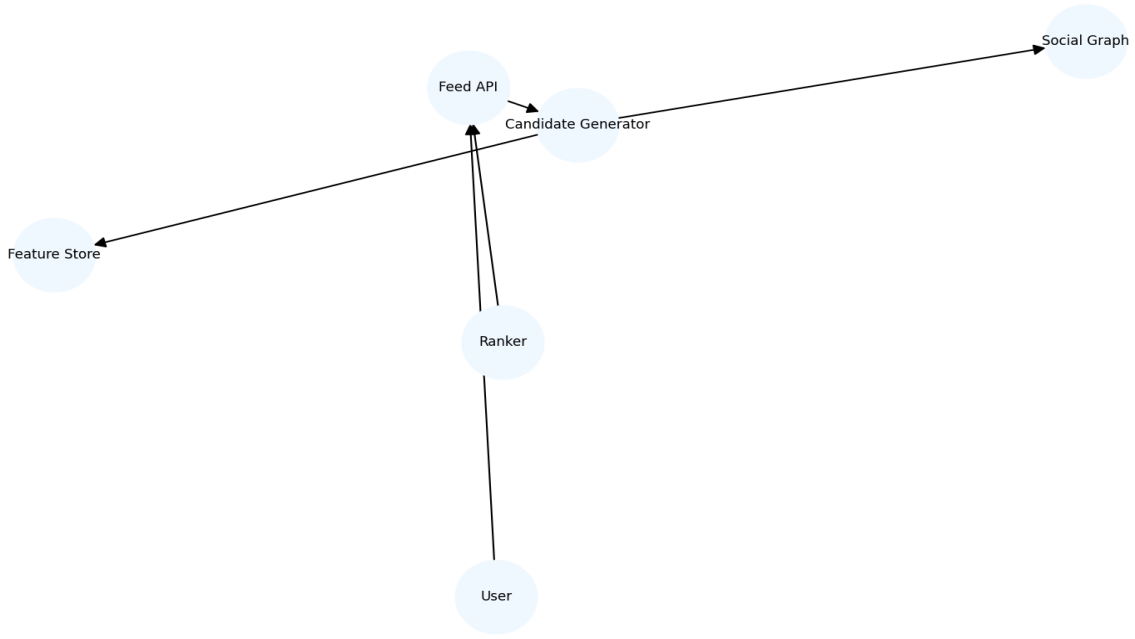
Show pipeline and explain indexing, retention, and alerting.



News Feed System (Facebook)

Assumptions: Billions of users, personalized ranking, low latency, freshness and relevance.
APIs: GET /feed?userId, POST /post, signals ingestion endpoints
Design: Signal collectors capture user actions -> features in feature store. Candidate generator (social graph, content-based) -> ranking service (ML model) -> personalization cache per user. Use fanout-on-write for low-fanout users and fanout-on-read hybrid for high-fanout.
Scaling: Partition feed by userId, precompute top candidates for heavy users, use edge caching and CDNs for media.
Failure Modes: Stale feeds: degrade to cached or heuristic feed. Hot users: rate-limit or prioritize.
Security: Privacy controls and ACL filtering.
Trade-offs: Fanout-on-write (fast reads, write amplification) vs fanout-on-read (slower reads, lower write cost).

Explain signal collection -> candidate generation -> ranking -> serving with caching strategies.



Messenger (Facebook Chat)

Assumptions: 1:1 and group chats, presence, typing indicators, message durability and optional E2E encryption.
APIs: WS /connect, POST /messages, GET /history
Design: Connection proxies for websockets, message routers partitioned by convo id, delivery workers, persistent logs, push gateways. Presence service scales separately.
Scaling: Shard by conversation id, regional proxies for low latency, store messages in append-only logs with compacted indexes.
Failure Modes: Offline delivery, message dedup, re-sync after reconnection.
Security: E2E optional, TLS, auth tokens.
Trade-offs: Server-side features vs strict E2E encryption limitations.

Walk connection -> send -> persist -> deliver -> ack flow.

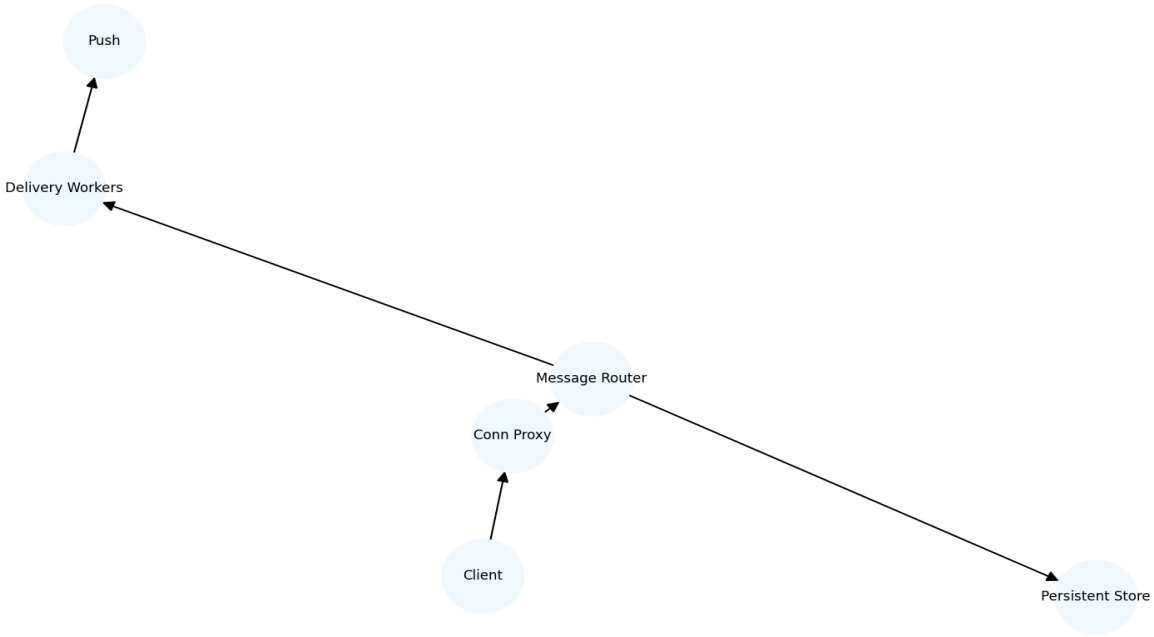
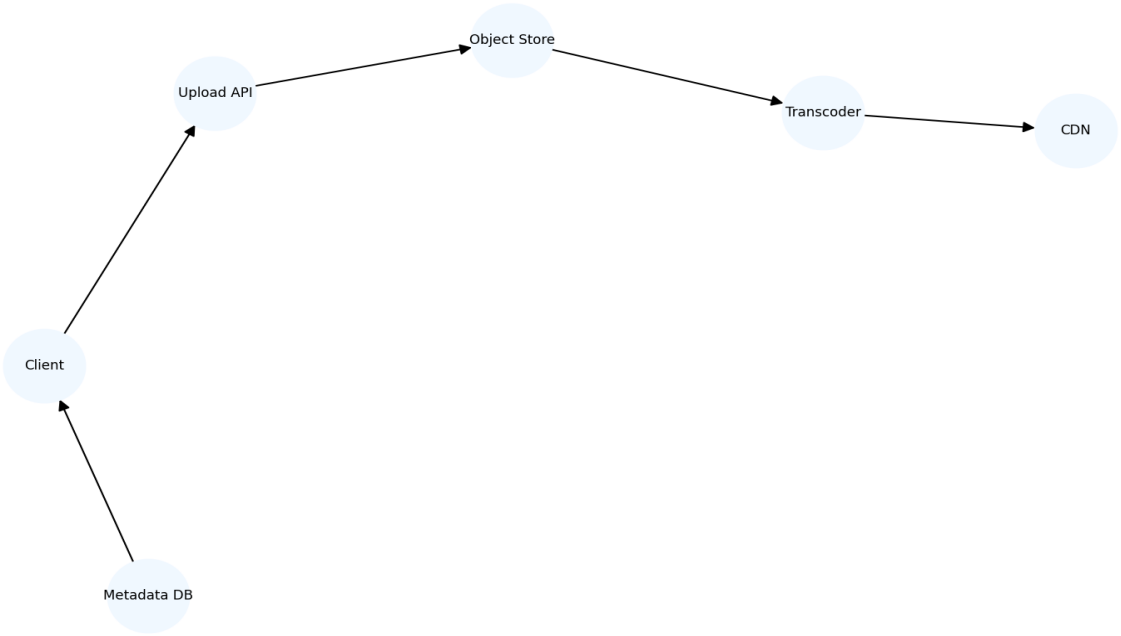


Photo Storage and Sharing

Assumptions: Billions of photos, user sharing, thumbnails/variants, metadata and tagging.
APIs: POST /upload, GET /photo/{id}?size=, POST /tag
Design: Ingest -> virus/quality checks -> store original in object storage -> generate variants via transcoding farm -> CDN for serving. Metadata in a scalable DB with indexes for tags and albums.
Scaling: Shard metadata by user or album, use per-object lifecycle policies, dedup via content hash.
Failure Modes: Corrupt uploads: validate and retry; CDN invalidation on updates.
Security: Access controls, share permissions, content moderation pipelines.
Trade-offs: Store originals vs optimized variants for cost vs quality.

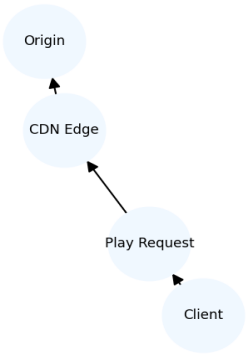
Describe upload to origin, variant generation, storage, CDN delivery, and moderation pipeline.

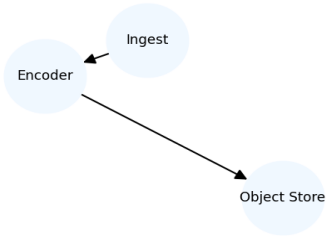


Video Platform (on-demand + live)

Assumptions: VOD and live, encoder farm, DRM possible, global delivery.
APIs: POST /upload, GET /play/{id}, live ingest endpoints
Design: Ingest -> encode to ABR ladders -> store segments in object storage -> CDN + origin shields -> playback clients with ABR and DRM. Live uses low-latency chunking and chunked transfer to edges.
Scaling: Per-title encoding, pre-warm popular streams, multi-CDN for resilience.
Failure Modes: Live latency spikes: use origin failover and multi-CDN; VOD: cache stampedes mitigated with origin shields.
Security: DRM, signed URLs, geo-restrictions.
Trade-offs: Startup latency vs segment size; CDN caching vs origin cost.

Explain VOD pipeline and differences for live streaming.

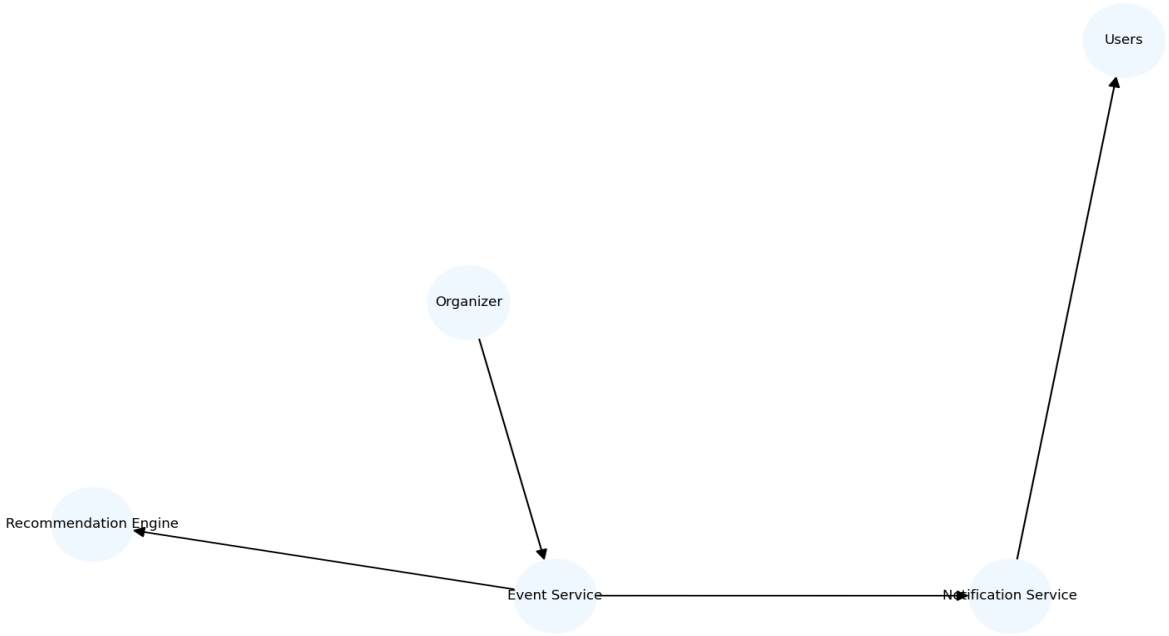




Event System (create/join/notify)

Assumptions: Users create events, RSVP, and receive notifications and recommendations.
APIs: POST /event, POST /rsvp, GET /events?userId
Design: Event service stores event metadata; notification service fans out invites; recommendation engine suggests events via signals; background jobs for reminders.
Scaling: Partition by organizer or region, rate-limit fanouts for large events, use batching for notifications.
Failure Modes: Missed notifications handled via retries; dedupe RSVP webhooks.
Security: Privacy settings and invite controls.
Trade-offs: Immediate notification vs batched digests for noise control.

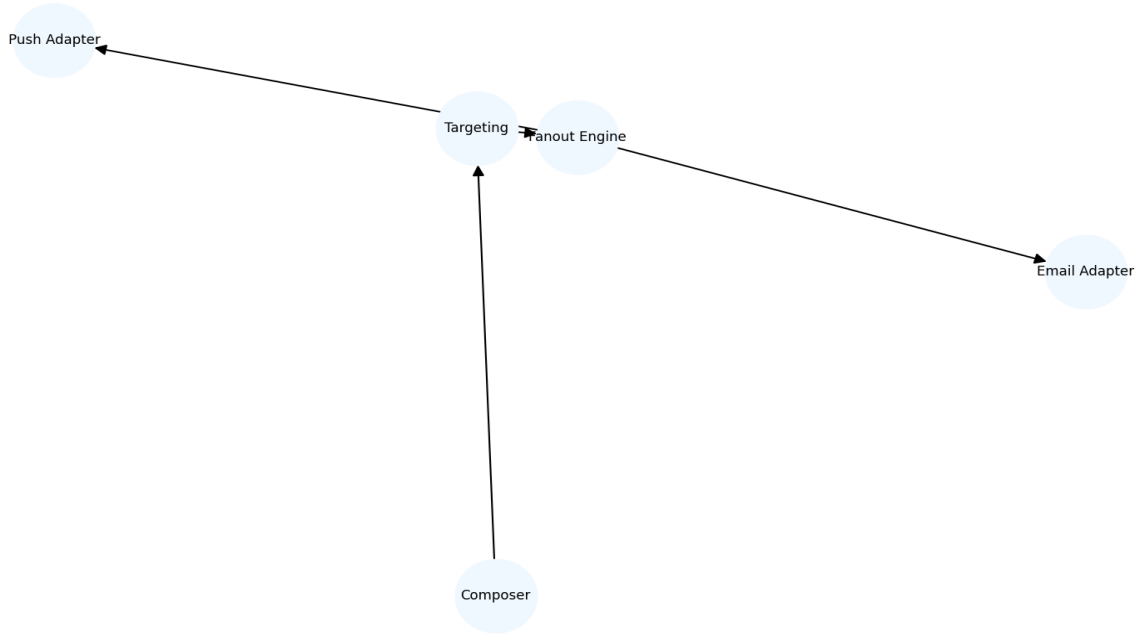
Walk event lifecycle from creation to notifications and recommendations.



Notification Service (fanout to billions)

Assumptions: High fanout, multi-channel (push, email, SMS), personalization.
APIs: POST /notify, subscription management APIs
Design: Notification composer -> targeting engine -> fanout engine which partitions recipients and uses per-channel adapters (push/email). Use priority queues and rate limits. Use per-user delivery logs for dedupe and retries.
Scaling: Shard by recipient hashes, batch sends, use third-party providers for deliverability.
Failure Modes: Provider failures: fallback to other providers; rate-limiting downstream.
Security: Consent and opt-outs, audit trails.
Trade-offs: Fanout immediacy vs cost; batching reduces cost but increases latency.

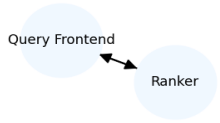
Explain targeting -> fanout -> per-channel adapters and retries.

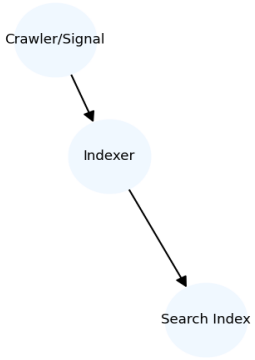


Search for People/Pages/Posts

Assumptions: High volume, many attributes, social graph signals, low-latency searches.
APIs: GET /search?q=, admin reindex APIs
Design: Indexing pipeline for users/pages/posts -> inverted indices with ranking signals, use multi-stage ranking (retrieval + re-rank with ML), personalization layer.
Scaling: Shard index by domain (users/pages/posts) and term ranges, warm caches for popular queries.
Failure Modes: Stale indexes, partial failures degrade to heuristics.
Security: ACLs for private content, safe-search for content.
Trade-offs: Indexing latency vs freshness; ranking cost vs throughput.

Walk crawl/index -> retrieval -> ranking -> serve path.





Fake Account / Spam Detection System

Assumptions: Large scale, need near real-time detection and batch offline analysis.
APIs: Signals ingestion, admin review APIs, quarantine actions
Design: Real-time stream processing for immediate signals (rate, IP, device), ML models for risk scoring, offline feature engineering and retraining, human review queue for edge cases. Use graph analysis for networked fraud.
Scaling: Feature pipelines partitioned by account id; scoring service must be low-latency at log scale.
Failure Modes: False positives: appeal workflows; evasion: continuous model updates.
Security: Data privacy, audit logs.
Trade-offs: Precision vs recall; stricter rules block good users but reduce spam.

Show streaming detection -> scoring -> action -> review loop.