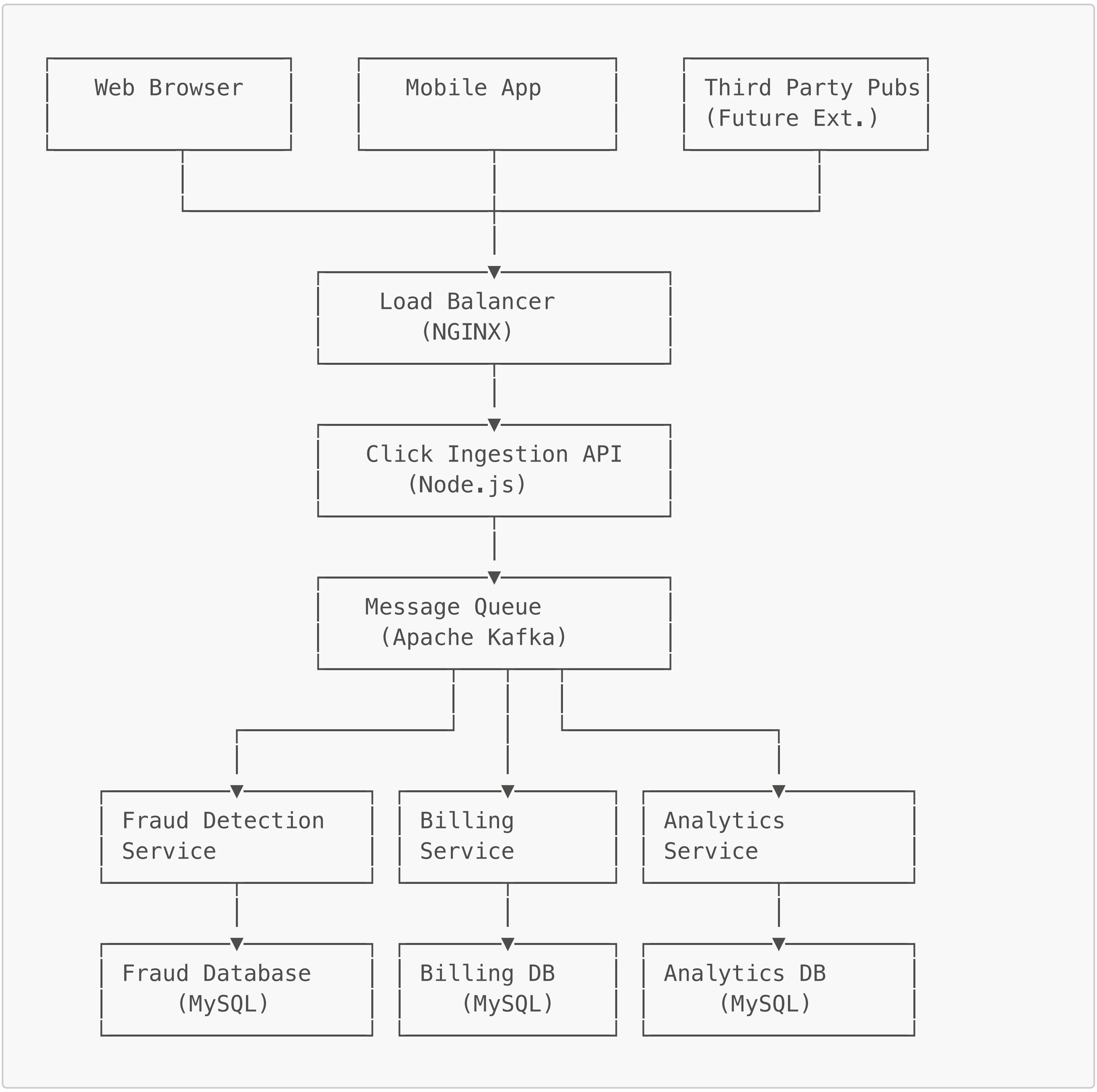
System Architecture of the Ad Click Charging System

High-Level Architecture Diagram



Component Details

1. Click Ingestion Layer

The Click Ingestion Layer consists of a load balancer and the click ingestion API. This layer is the first layer where the clicks made on the ads will enter the system.

- **Load Balancer**: We need a load balance layer that can handle concurrent connections and distribute the load (i.e. incoming clicks) across multiple instances, there by protecting the system from overload. NGINX is one of the light weight options widely used in the industry.

- **Click Ingestion API**: This is a simple HHTP server. This server validates the incoming clicks and decorates it with additional information, in our case time stamp and ip address, and passes the validated clicks to the next stage of processing. Node.js would be a good option to clickly implement a simple HTTP server.

- **Rate Limiting**: Basic rate limiting to prevent Denial of Service attacks. Ideally the load balancer or the API Gateay would provide it. For a simple setup, it is set to 1000 clicks/minute/IP address for this project.

2. Message Queue (Pub-Sub Core)

Heart of the pub-sub architecture. Using Kafka for pub-sub

- **Apache Kafka**: Selected Kafka over Redis for messaging because it can handle much higher throughput and has built-in persistence. So, if the service goes down, we won't lose any click events.

- **Topics Structure**: Designed three main topics:
  - `click-events`: Raw clicks from the API (before any processing)
  - `validated-clicks`: Clicks that cleared fraud detection
  - `billing-events`: Clicks that were successfully charged

- **Publisher/Subscriber Pattern**: The ingestion API publishes to click-events, then each service subscribes to what it needs. This way if we want to add a new service later (eg reporting), just need to subscribe it to the right topics.

3. Processing Services

Split the processing into three separate services as per microservices architecture:

- **Fraud Detection Service**: This is the most complex component in a production system. However for the current project, we habe implemenetd a simple fraud detection system based on the velocity and location. The logic includes validating user agent, velocity of incoming clicks and locations from where the clicks originated i.e. the IP address. These simple checks could handle a wide variety of fraudulent clicks, though it will fall well short of sophisticated attacks.

- **Billing Service**: This service handles the money side of things. It calculates the costs to an advertiser based on campaign bid amounts and accordingly reduces available advertiser budget. For the implementation of billing service, we use transation as we are dealing with financial data and we would like to be extra careful with financials.

- **Analytics Service**: This service collects real-time metrics. This will subscribes to billing events and create hourly/daily aggregations for reporting. We should note here that data collected by Analytics service is not used for billing. It is rather downstream to billing

4. Data Layer

We keep the the database architecture simple and use MySQL for all usecases. The three usecases we implement are Fraud, Billing and Analytics.

- **Fraud Database**: This stores the result of click validation and the fraud scores for each click

- **Billing Database**: This stores the financial transactions, advertiser budgets and the campaign budgets
- **Analytics Database**: This stores the time-series data, which are pre-calculated metrics
- **Redis Cache**: We used Redic as the cache for the session data and the counter for rate limits. We use cache as it is much faster than using MySQL for these operations which are frequenct in nature.

Design Decisions and Challenges

Why Pub-Sub Architecture

Selected pub-sub architecture for the following reasons:

- **Decoupling**: Each service operates independently. If the analytics service crashes, billing continues to work.
- **Scalability**: Can add more instances of any service without modifying others.
- **Flexibility**: Adding new features requires only subscribing to the appropriate topics.

Implementation Challenges

- **Message Ordering**: Addressed out-of-order processing by using Kafka partitioning with campaign_id as the key.
- **Exactly-Once Processing**: Prevented duplicate charges by implementing idempotency checks using unique event IDs.
- **Database Performance**: Added indexes and implemented connection pooling to handle high click volumes.

Performance Targets

Based on research of production ad platforms:

- 10,000 clicks per second throughput
- Under 100ms API response time
- Under 50ms fraud detection processing
- 99.9% system availability

The architecture is designed to support these targets with proper optimization and tuning.

Implementation Status

Completed Components

- Click ingestion and validation
- Kafka message publishing and consuming
- Basic fraud detection for bot traffic
- Billing calculations with budget controls
- Real-time analytics aggregation

Remaining Work

- Enhanced fraud detection models
- Comprehensive load testing
- Monitoring dashboard implementation
- Improved error handling and recovery mechanisms
- Security enhancements (authentication, encryption)