A/B Testing & Experimentation Platform - High-Level Design (HLD)

Table of Contents

Problem Statement

Design an A/B testing and experimentation platform like Optimizely, Google Optimize, or LaunchDarkly that allows companies to:

- Create experiments with multiple variants
- Assign users to variants consistently
- Track user behavior and conversions
- Analyze experiment results with statistical significance
- Gradually roll out features (feature flags)
- Make data-driven decisions

Example Use Case

```
E-commerce company wants to test new checkout flow:
- Variant A (Control): Current checkout (50% of users)
- Variant B: Single-page checkout (50% of users)

Goal: Which variant has higher conversion rate?

Platform should:
1. Consistently assign users to same variant
2. Track conversions for each variant
3. Calculate statistical significance
4. Determine winning variant
```

Scale Requirements

- **10,000+ experiments** running concurrently
- **1 billion events** tracked per day
- **100 million users** actively participating in experiments
- **< 10ms latency** for variant assignment
- **Real-time analytics** for experiment monitoring

Functional Requirements

Must Have (P0)

1. **Experiment Management**
    - Create experiments with multiple variants
    - Define traffic allocation (50-50, 70-30, etc.)
    - Set start and end dates
    - Pause/resume experiments
    - Target specific user segments
2. **User Assignment**
    - Consistent assignment (same user → same variant)
    - Fast assignment (< 10ms)
    - Handle billions of users
    - Support override for testing
    - Exclude bots/internal traffic
3. **Event Tracking**
    - Track user actions (clicks, purchases, signups)
    - Support custom events
    - Handle billions of events/day
    - Associate events with experiments
    - Real-time event ingestion

4. **Analytics & Reporting**
    - Conversion rates per variant
    - Statistical significance calculation
    - Confidence intervals
    - Real-time dashboards
    - Historical data analysis
    - Export reports
5. **Feature Flags**
    - Gradually roll out features (0% → 10% → 50% → 100%)
    - Target specific users/segments
    - Kill switch (instant rollback)
    - Percentage-based rollouts

Nice to Have (P1)

- Multi-variate testing (test multiple variables)
- Holdout groups (control group across experiments)
- Interaction effects (experiment A affects experiment B)
- Sequential testing (early stopping)
- Revenue impact calculation
- Experiment recommendations (what to test next)
- Integration with analytics tools (Google Analytics, Mixpanel)

Non-Functional Requirements

Performance

- **Variant assignment**: < 10ms for p99
- **Event ingestion**: < 100ms
- **Dashboard load**: < 1 second
- **Report generation**: < 5 seconds for real-time, < 1 hour for historical

Scalability

- Handle 100M active users
- Process 1B events per day (12K QPS)
- Support 10K concurrent experiments
- Scale to 1000+ companies using platform

Availability

- **99.99% uptime** for assignment service (critical path)
- **99.9% uptime** for analytics service
- No single point of failure
- Graceful degradation

Consistency

- **Strong consistency** for experiment configuration (can't show wrong variant)
- **Eventual consistency** for analytics (acceptable delay < 1 minute)
- **Deterministic** assignment (same user always gets same variant)

Security

- User privacy (GDPR, CCPA compliance)
- Secure experiment configuration
- Access control (who can create/view experiments)
- Audit logs for all changes

Capacity Estimation

Traffic Estimates

```
Active Experiments: 10,000
Users in experiments: 100M
Events per user per day: 10 (average)

Assignment Requests:
- Users × Experiments × Checks = 100M × 10 × 10 = 10B/day
- QPS: 10B / 86,400 ≈ 115K QPS
- Peak (3x): ~350K QPS

Event Tracking:
- Events: 100M × 10 = 1B/day
- QPS: 1B / 86,400 ≈ 12K QPS
- Peak: ~35K QPS

Read-to-Write Ratio:
- Assignments (reads): 115K QPS
- Events (writes): 12K QPS
- Ratio: ~10:1 (read-heavy)
```

Storage Estimates

```
Experiment Configurations:
- 10,000 active experiments
- 1 KB per experiment config
- Total: 10 MB (tiny!)

User Assignments:
- 100M users × 10 experiments average = 1B assignments
- Assignment record: 50 bytes (user_id, experiment_id, variant_id)
- Total: 1B × 50 bytes = 50 GB
```

```
Event Data:
- 1B events per day
- Event record: 200 bytes (user_id, experiment_id, event_type,
timestamp, metadata)
- Daily: 1B × 200 bytes = 200 GB
- Yearly: 200 GB × 365 = 73 TB
- 5 years: 365 TB

Aggregated Metrics:
- Per experiment per variant per day
- 10K experiments × 3 variants × 365 days = 11M records
- 1 KB per record = 11 GB per year
- 5 years: 55 GB

Total Storage (5 years): ~365 TB
```

Bandwidth Estimates

```
Incoming (Event tracking):
- 12K QPS × 200 bytes = 2.4 MB/s

Outgoing (Variant assignment):
- 115K QPS × 50 bytes = 5.75 MB/s

Dashboard queries:
- Negligible compared to assignment traffic

Total: ~8 MB/s (manageable)
```

Memory Estimates (Caching)

```
Cache all active experiment configs:
- 10K experiments × 1 KB = 10 MB (cache everything!)

Cache user assignments (hot users, 20%):
- 20M users × 10 experiments × 50 bytes = 10 GB

Cache aggregated metrics (24 hours):
- 10K experiments × 3 variants × 1 KB = 30 MB

Total cache: ~11 GB (single Redis instance!)
```

High-Level Architecture

```
                        [Users/Applications]
                                │
                                ↓
                      [CDN - CloudFront]
                      (Dashboard, Reports)
                                │
                                ↓
                      [DNS - Route53]
                                │
                                ↓
                  [Load Balancer - ALB]
                                │
                  ┌─────────────┴─────────────┐
                  ↓                           ↓
          [API Gateway]              [WebSocket Server]
       (Auth, Rate Limiting)         (Real-time Updates)
                  │
      ┌───────────┼───────────┬───────────────┐
      ↓           ↓           ↓               ↓
[Assignment  [Event      [Analytics    [Admin
 Service]    Service]    Service]      Service]
(Get variant) (Track)    (Reports)    (Manage exps)
      │           │           │               │
      └───────────┼───────────┴───────────────┘
                  ↓
          [Cache Layer - Redis]
       (Experiment configs, assignments)
                  │
      ┌───────────┴──────────┬───────────────┐
      ↓                      ↓               ↓
[Config DB]          [Event Stream]    [Analytics DB]
(PostgreSQL)            (Kafka)        (ClickHouse/
(Experiments,                          Cassandra)
 Assignments)
      │                      │               │
      └──────────────────────┴───────────────┘

                  ↓
          [Object Storage - S3]
          (Raw events, exports)
                  ↓
      [Data Processing Pipeline]
         (Spark/Flink)
                  ↓
      [Data Warehouse - Redshift]
      (Historical analysis)
```

Core Components

1. Assignment Service

Purpose: Determine which variant user should see

Responsibilities:

- Fetch experiment configuration
- Check if user eligible for experiment
- Assign user to variant (deterministic)
- Return variant details
- Cache assignments

Critical: This is on critical path (every user request)

- Must be fast (< 10ms)
- Must be highly available (99.99%)
- Must be deterministic (same user → same variant)

Why Go or Java over alternatives?

Feature	Go	Java (Spring Boot)	Node.js	Python
Performance	⚡ Very Fast	⚡ Fast	🐢 Moderate	🐌 Slow
Concurrency	✅ Goroutines (native)	✅ Threads	⚠ Event loop	⚠ Limited (GIL)
Memory	✅ Low footprint	🐢 High (JVM)	✅ Moderate	✅ Moderate
Latency	✅ Consistent (~1ms)	✅ Good (~2-3ms)	⚠ Variable	🐌 Higher
Type Safety	✅ Compile-time	✅ Compile-time	❌ Runtime	⚠ Optional
Ecosystem	⚠ Growing	✅ Very mature	✅ Mature	✅ Mature
Deployment	✅ Single binary	🐢 JAR + JVM	✅ Easy	✅ Easy

Decision: Go (primary) or Java (acceptable)

Why Go:

1. **Raw performance**: Hash calculation + Redis lookup in ~1-2ms
2. **Low memory**: 10-20 MB per service instance vs 100-500 MB for Java
3. **Fast startup**: <1 second vs 5-10 seconds for Java
4. **Native concurrency**: Goroutines handle 115K QPS easily
5. **Single binary**: Easy deployment, no dependency hell

Why Java is also good:

1. **Mature ecosystem**: Battle-tested libraries, frameworks
2. **Team expertise**: More Java developers available
3. **Spring Boot**: Rapid development
4. **Debugging tools**: Excellent profiling, monitoring

Why NOT Node.js:

- Event loop can block on CPU-intensive hashing
- Variable latency (GC pauses unpredictable)
- Better for I/O-bound, not CPU-bound tasks

Why NOT Python:

- Too slow for < 10ms requirement
- GIL limits concurrency
- Great for analytics, poor for high-performance services

Code Example (Java):

```java
public class AssignmentService {
    private final RedisClient redis;
    private final MessageDigest md5;

    public AssignmentService(RedisClient redis) {
        this.redis = redis;
        try {
            this.md5 = MessageDigest.getInstance("MD5");
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * Assign user to variant with caching
     * Time Complexity: O(1) with cache, O(v) without
     * Expected Latency: <2ms with cache, ~3ms without
     */
    public Variant assignUser(String userId, String experimentId, 
Experiment exp) {
        String cacheKey = userId + ":" + experimentId;

        // Check Redis cache (< 1ms)
        Variant cached = redis.get(cacheKey, Variant.class);
        if (cached != null) {
            return cached;  // Cache hit - fast path
        }

        // Calculate deterministic hash (< 1ms)
        int bucket = calculateBucket(userId, experimentId, 
exp.getSalt());

        // Find variant by bucket (< 1ms)
        Variant variant = findVariantByBucket(bucket, 
exp.getVariants());

        // Store in cache with TTL (< 1ms)
        redis.setex(cacheKey, exp.getDurationSeconds(), variant);
        // Async write to PostgreSQL for audit (non-blocking)
        CompletableFuture.runAsync(() -> {
            saveAssignmentToDatabase(userId, experimentId, 
variant.getId());
        });

        return variant;  // Total: ~2-3ms
    }

    private int calculateBucket(String userId, String experimentId, 
String salt) {
        String input = userId + ":" + experimentId + ":" + salt;
        byte[] digest = md5.digest(input.getBytes());
        BigInteger bigInt = new BigInteger(1, digest);
        return Math.abs(bigInt.intValue() % 100);  // 0-99
    }

    private Variant findVariantByBucket(int bucket, List<Variant> 
variants) {
        int rangeStart = 0;
        for (Variant v : variants) {
            if (bucket >= rangeStart && bucket < rangeStart + 
v.getTrafficPercentage()) {
                return v;
            }
            rangeStart += v.getTrafficPercentage();
        }
        return variants.get(0);  // Fallback
    }
}
```

Performance Comparison:

```
Java:    2-3ms per assignment ✅ (Chosen for production)
Go:      1-2ms per assignment (Alternative)
Node.js: 3-5ms per assignment
Python:  5-10ms per assignment

At 115K QPS, Java provides excellent performance with mature ecosystem
```

---

2. Event Tracking Service (Detailed)

Purpose: Capture user actions and experiment exposure at scale

Critical Requirements:

- **Non-blocking**: Must not slow down user experience
- **High throughput**: Handle 12K-35K events/sec

- **Reliability**: No event loss (durability)
- **Low latency**: Return response in < 100ms
- **Batching**: Efficient bulk processing

Architecture:

```
[Client SDK] → [Event Tracking API] → [Validation] → [Kafka] →
[Consumers]
   (Async)        (Fast response)       (Enrich)    (Buffer)   (Process)
```

Detailed Flow:

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 1: Client-Side Event Collection                        │
└─────────────────────────────────────────────────────────────┘
Client SDK (JavaScript/Mobile):
- Buffer events locally (in-memory queue)
- Batch 10-50 events together
- Send async (doesn't block UI)
- Retry on failure with exponential backoff

┌─────────────────────────────────────────────────────────────┐
│ STEP 2: API Reception (Event Tracking Service)              │
└─────────────────────────────────────────────────────────────┘
Event Service receives batch:
- Authenticate request (API key)
- Rate limit check (per client)
- Quick validation (schema, required fields)
- Return 202 Accepted immediately (<10ms)
- Process async in background

┌─────────────────────────────────────────────────────────────┐
│ STEP 3: Event Enrichment & Validation                       │
└─────────────────────────────────────────────────────────────┘
Enrich each event with:
- Server timestamp (accurate time)
- IP address → GeoIP (country, city)
- User agent → Device info (browser, OS)
- Session ID (if not provided)
- Experiment/Variant validation

Validation:
- Experiment exists and is active
- Variant belongs to experiment
- Event type is valid
- Required fields present

┌─────────────────────────────────────────────────────────────┐
│ STEP 4: Write to Kafka                                      │
```

```
  └────────────────────────────────────────────────────────────┘
Batch write to Kafka:
- Topic selection based on event type
- Partition by user_id (maintain ordering)
- Compression (LZ4 or Snappy)
- Async acknowledge

  ┌────────────────────────────────────────────────────────────┐
  │ STEP 5: Multiple Consumers Process Events                  │
  └────────────────────────────────────────────────────────────┘
Consumer 1 - Flink (Real-time):
- Aggregate metrics every 1 minute
- Update ClickHouse

Consumer 2 - Spark (Batch):
- Hourly batch processing
- Complex analytics

Consumer 3 - Warehouse:
- Store in Redshift for historical analysis

Consumer 4 - Monitoring:
- Alert on anomalies
```

Java Implementation:

```java
@RestController
@RequestMapping("/api/v1")
public class EventTrackingController {

    private final EventTrackingService eventService;
    private final RateLimiter rateLimiter;

    @PostMapping("/track")
    public ResponseEntity<TrackResponse> trackEvent(@RequestBody
TrackRequest request) {
        // Rate limiting (per client)
        if (!rateLimiter.allowRequest(request.getClientId())) {
            return ResponseEntity.status(429).build();  // Too Many
Requests
        }

        // Quick validation
        if (!isValid(request)) {
            return ResponseEntity.badRequest().build();
        }

        // Process async (non-blocking)
        CompletableFuture.runAsync(() -> {
            eventService.processEvent(request);
        });
        // Return immediately
        return ResponseEntity.accepted()
            .body(new TrackResponse("success", generateEventId()));
    }

    @PostMapping("/track/batch")
    public ResponseEntity<BatchTrackResponse> trackBatch(
            @RequestBody BatchTrackRequest request) {

        // Validate batch
        if (request.getEvents().size() > 100) {
            return ResponseEntity.badRequest()
                .body(new BatchTrackResponse("error", "Max 100 events
per batch"));
        }

        // Process all events async
        CompletableFuture.runAsync(() -> {
            for (TrackRequest event : request.getEvents()) {
                eventService.processEvent(event);
            }
        });

        return ResponseEntity.accepted()
            .body(new BatchTrackResponse("success",
request.getEvents().size()));
    }
}

@Service
public class EventTrackingService {

    private final KafkaTemplate<String, Event> kafkaTemplate;
    private final GeoIPService geoIPService;
    private final ExperimentConfigCache configCache;

    /**
     * Process single event
     *
     * Time Complexity: O(1) for validation + O(1) for Kafka write
     * Expected Latency: 2-5ms
     */
    public void processEvent(TrackRequest request) {
        try {
            // Create event object
            Event event = new Event();
            event.setEventId(UUID.randomUUID().toString());
            event.setUserId(request.getUserId());
            event.setExperimentId(request.getExperimentId());
            event.setVariantId(request.getVariantId());
            event.setEventType(request.getEventType());
            event.setEventName(request.getEventName());
            event.setTimestamp(Instant.now());

            // Enrich with metadata
            enrichEvent(event, request);

            // Validate experiment/variant exists
            if (!validateExperiment(event)) {
                logInvalidEvent(event, "Invalid experiment or variant");
                return;
            }

            // Write to Kafka (async)
            String topic = selectTopic(event.getEventType());
            String key = event.getUserId();  // Partition by user

            kafkaTemplate.send(topic, key, event)
                .addCallback(
                    success -> logSuccess(event),
                    failure -> handleFailure(event, failure)
                );

        } catch (Exception e) {
            logError("Event processing failed", e);
            // Don't throw - we don't want to fail the API call
        }
    }

    /**
     * Enrich event with additional metadata
     */
    private void enrichEvent(Event event, TrackRequest request) {
        // GeoIP lookup from IP address
        String ipAddress = request.getIpAddress();
        if (ipAddress != null) {
            GeoLocation geo = geoIPService.lookup(ipAddress);
            event.setCountry(geo.getCountry());
            event.setCity(geo.getCity());
            event.setRegion(geo.getRegion());
        }

        // Parse user agent
        String userAgent = request.getUserAgent();
        if (userAgent != null) {
            UserAgentInfo uaInfo = parseUserAgent(userAgent);
            event.setBrowser(uaInfo.getBrowser());
            event.setOs(uaInfo.getOs());
- Want self-hosted solution

**Trade-offs:**
- PostgreSQL requires more operational overhead than DynamoDB
- But we need the query flexibility and relationships

**Schema**:
```

Experiments:

- experiment_id, name, description
- start_date, end_date, status
- hypothesis, target_metric
- created_by, updated_at

Variants:

- variant_id, experiment_id, name
- traffic_allocation (percentage)
- configuration (JSON)

User_Assignments:

- user_id, experiment_id, variant_id
- assigned_at, sticky (boolean)
- INDEX on (user_id, experiment_id)

Targeting_Rules:

- rule_id, experiment_id
- dimension (country, platform, user_segment)
- operator (equals, in, greater_than)
- values

```


---

### 6. Event Stream (Kafka)

**Purpose**: Durable, scalable event pipeline

**Topics**:
- `experiment-exposures`: User saw variant
- `experiment-conversions`: User completed goal
- `custom-events`: Custom tracking events

**Why Kafka over alternatives?**

| Feature | Apache Kafka | RabbitMQ | Amazon SQS | Google Pub/Sub |
|---------|-------------|----------|------------|----------------|
| **Throughput** | ⚡ Very High (millions/sec) | 🐢 Moderate (50K/sec) |
🐢 Moderate | ⚡ High |
| **Durability** | ✅ Disk-backed, replicated | ⚠ Memory-first | ✅
Managed | ✅ Managed |
| **Ordering** | ✅ Per-partition | ⚠ Per-queue | ⚠ Best-effort | ⚠
Best-effort |
| **Replay** | ✅ Yes (time-based) | ❌ No | ❌ No | ❌ No |
| **Multiple Consumers** | ✅ Independent | ⚠ Competing | ⚠ Competing |
✅ Independent |
| **Retention** | ✅ Days to weeks | ⚠ Until consumed | ⚠ 14 days max |
⚠ 7 days max |
| **Latency** | Low (ms) | Very Low (ms) | Moderate (ms) | Low (ms) |
| **Cost** | 💰 Medium (self-hosted) | 💰 Low | 💰💰 Pay per request |
💰💰 Pay per message |

**Decision: Apache Kafka**

**Reasons:**
1. **High throughput**: Handles 12K-35K events/sec easily
   - Kafka designed for 100K+ messages/sec per broker
   - Our peak 35K QPS is comfortable for 3-broker cluster

2. **Multiple independent consumers**: Critical requirement
```

Single event stream → Multiple consumers:

- Flink (real-time aggregation)
- Spark (batch processing)
- Data warehouse (historical storage)
- Monitoring system (alerting)
- Audit log (compliance)

Each consumes at own pace without affecting others

```


3. **Replay capability**: Essential for debugging
- Bug in analytics? Replay events from yesterday
- New metric added? Reprocess historical events
- Data correction? Recompute from source

4. **Durability**: Events replicated across 3 brokers
- Can lose 2 brokers and still have data
- Disk-backed (survives restarts)
- No data loss

5. **Ordering guarantees**: Per partition
- User's events in order (use user_id as key)
- Important for funnel analysis, session reconstruction

**Why NOT RabbitMQ:**
- Lower throughput (50K vs millions/sec)
- No replay capability (events deleted after consumption)
- Memory-first (lose data on crash)
- Better for task queues, not event streaming

**Why NOT Amazon SQS:**
- No replay (events deleted after consumption)
- Best-effort ordering (not guaranteed)
- Higher cost at scale (pay per million requests)
- We process 1B events/day = 1000 million requests = $$

**Why NOT Google Pub/Sub:**
- Excellent alternative, similar to Kafka
- Vendor lock-in (Google Cloud only)
- Higher cost (pay per message + data transfer)
- We prefer self-hosted solution

**Trade-offs:**
- Kafka more complex to operate than managed services
- But replay capability and throughput justify the complexity
- Cost savings: ~$1,500/month vs $10,000+ for managed at scale

---

### 8. Cache Layer (Redis)

**Purpose**: Ultra-fast experiment configuration and assignment lookup

**Why Redis over alternatives?**

| Feature | Redis | Memcached | Hazelcast | Apache Ignite |
|---------|-------|-----------|-----------|---------------|
| **Speed** | ⚡ Sub-ms | ⚡ Sub-ms | 🐢 Few ms | 🐢 Few ms |
| **Data Structures** | ✅ Rich (hash, set, sorted set) | ⚠ Key-value only | ✅ Rich | ✅ Rich |
| **Persistence** | ✅ Optional (RDB, AOF) | ❌ None | ⚠ Limited | ✅ Yes |
| **Replication** | ✅ Master-slave, cluster | ❌ Client-side | ✅ Yes | ✅ Yes |
| **Pub/Sub** | ✅ Built-in | ❌ No | ⚠ Limited | ✅ Yes |
| **TTL** | ✅ Per-key | ✅ Per-key | ✅ Yes | ✅ Yes |
| **Memory Efficiency** | ✅ Good | ✅ Better | 🐢 Higher overhead | 🐢 Higher overhead |
| **Maturity** | ✅ Very mature | ✅ Mature | ⚠ Less mature | ⚠ Less mature |

**Decision: Redis**

**Reasons:**
1. **Sub-millisecond latency**: Critical for < 10ms requirement
- Redis GET: 0.1-1ms
- Memcached GET: 0.1-1ms
- Hazelcast GET: 2-5ms
- With < 10ms budget, every ms counts

2. **Rich data structures**: Flexible caching
```

Strings: Cache variant IDs
Hashes: Cache experiment configs (nested data)
Sets: Cache user segments, excluded users
Sorted Sets: Cache experiment priorities

```
3. **Persistence optional**: Can survive restarts
- RDB snapshots (point-in-time)
- AOF (append-only log)
- Acceptable to lose cache (rebuild from DB)

4. **Redis Cluster**: No single point of failure
- Automatic sharding across nodes
- Master-slave replication
- Automatic failover
- Handles 115K QPS across 3-6 nodes

5. **All experiment configs fit in memory**: 10 MB
- Redis handles TB-scale, we only need GB
- Can cache everything (100% hit ratio)

**Why NOT Memcached:**
- Memcached would work for simple key-value
- Redis has richer data structures (we use hashes)
- Redis has persistence (nice to have)
- Redis has built-in pub/sub (useful for cache invalidation)
- Similar performance, Redis more flexible

**Why NOT Hazelcast:**
- Hazelcast is in-process (embedded in app)
- Higher memory overhead (JVM-based)
- More complex than needed
- Redis simpler and faster

**Why NOT Apache Ignite:**
- Too heavy for our use case
- We don't need compute grid features
- Redis sufficient and simpler

**Cache Strategy:**
```

Write-Through for Experiment Configs:

- Create experiment → Write to PostgreSQL → Update Redis
- Ensures cache always fresh

Cache-Aside for User Assignments:

- Check Redis first
- If miss: Calculate → Store in Redis

- 99.9% hit ratio (almost always in cache)

Cache Invalidation:

- TTL: 5 minutes (configs)
- Active invalidation: On experiment update
- Pub/Sub: Notify all servers of changes

```
**Performance with Redis:**
```

Without Cache (PostgreSQL only):

- 115K QPS × 5ms per query = 575 seconds of DB time/sec
- Need 575 database connections (impossible!)
- Database bottleneck

With Redis Cache:

- 99.9% hit ratio
- 115K QPS × 1ms (Redis) = 115 connections equivalent
- 0.1% miss: 115 QPS × 5ms (PostgreSQL) = manageable
- Database load reduced 1000x

Result: < 10ms latency achieved!

```
**What to Cache:**
```

Experiment Configs:
Key: exp:config:{experiment_id}
Value: {variants, allocation, rules}
TTL: 5 minutes

User Assignments:
Key: assign:{user_id}:{experiment_id}
Value: variant_id
TTL: Duration of experiment

Aggregated Metrics (hourly):
Key: metrics:{experiment_id}:{variant_id}:{hour}
Value: {exposures, conversions, rate}
TTL: 7 days

```
**Why Critical**:
```

```
- 115K assignment QPS → Can't hit database
- < 10ms latency requirement
- 99.99% cache hit ratio possible

---

### 7. Analytics Database (ClickHouse/Cassandra)

**Purpose**: Fast analytical queries on billions of events

**Why ClickHouse**:
- Columnar storage (fast aggregations)
- Designed for OLAP
- Real-time queries
- Handles billions of rows

**Alternative: Cassandra**
- Better for write-heavy
- Time-series optimization
- Geographic distribution

**Schema** (ClickHouse):
```

events:

- timestamp
- user_id
- experiment_id
- variant_id
- event_type
- properties (JSON)
- country, platform, device

Partitioned by date
Optimized for aggregate queries

```


---

### 8. Cache Layer (Redis)

**Purpose**: Ultra-fast experiment configuration and assignment lookup

**What to Cache**:
```

Experiment Configs:
Key: exp:config:{experiment_id}

Value: {variants, allocation, rules}
TTL: 5 minutes

User Assignments:
Key: assign:{user_id}:{experiment_id}
Value: variant_id
TTL: Duration of experiment

Aggregated Metrics (hourly):
Key: metrics:{experiment_id}:{variant_id}:{hour}
Value: {exposures, conversions, rate}
TTL: 7 days

```


**Why Critical**:
- 115K assignment QPS → Can't hit database
- < 10ms latency requirement
- 99.99% cache hit ratio possible

---

## API Design

### REST API Endpoints

#### Assignment APIs (Client-facing)
```

POST /api/v1/assign
Request:
{
"user_id": "user_123",
"experiments": ["checkout_test", "homepage_redesign"],
"context": {
"country": "US",
"platform": "web",
"user_segment": "premium"
}
}

Response:
{
"assignments": [
{
"experiment_id": "checkout_test",
"variant_id": "variant_b",
"variant_name": "single_page_checkout",
"config": {"button_color": "blue", "layout": "compact"}

},
{
"experiment_id": "homepage_redesign",
"variant_id": "variant_a",
"variant_name": "control"
}
]
}

```
#### Event Tracking APIs
```

POST /api/v1/track
Request:
{
"user_id": "user_123",
"experiment_id": "checkout_test",
"variant_id": "variant_b",
"event_type": "conversion",
"event_name": "purchase_completed",
"properties": {
"revenue": 99.99,
"currency": "USD",
"items": 3
},
"timestamp": "2025-01-08T10:00:00Z"
}

Response:
{
"success": true,
"event_id": "evt_abc123"
}

```
#### Experiment Management APIs
```

POST /api/v1/experiments
GET /api/v1/experiments/{id}
PUT /api/v1/experiments/{id}
DELETE /api/v1/experiments/{id}
POST /api/v1/experiments/{id}/start
POST /api/v1/experiments/{id}/stop

GET /api/v1/experiments/{id}/results
GET /api/v1/experiments/{id}/realtime

```
#### Analytics APIs
```

GET /api/v1/analytics/experiment/{id}
Response:
{
"experiment_id": "checkout_test",
"status": "running",
"duration_days": 14,
"sample_size": 50000,
"variants": [
{
"variant_id": "variant_a",
"name": "control",
"exposures": 25000,
"conversions": 2500,
"conversion_rate": 10.0,
"confidence_interval": [9.2, 10.8]
},
{
"variant_id": "variant_b",
"name": "treatment",
"exposures": 25000,
"conversions": 2875,
"conversion_rate": 11.5,
"confidence_interval": [10.7, 12.3],
"lift": "+15%",
"statistical_significance": 0.95,
"p_value": 0.003
}
],
"recommendation": "variant_b wins with 95% confidence"
}

```
---

## Deep Dives

### 1. User Assignment Algorithm (The Core Challenge)

**Requirements for Assignment**:
```

```
```

1. Deterministic: Same user always gets same variant
2. Random: 50-50 split should be truly 50-50
3. Fast: < 10ms latency
4. Scalable: Handle billions of users
5. Consistent: Even if system restarts

```


---

#### Approach 1: Hash-Based Assignment (Recommended)

**How It Works**:
```

Assignment = hash(user_id + experiment_id + salt) % 100

If Assignment < 50: Variant A (50%)
If Assignment >= 50: Variant B (50%)

Properties:

- Deterministic (same inputs → same output)
- Random distribution (hash function ensures uniformity)
- Fast (O(1) computation)
- Stateless (no database lookup needed)

```


**Example**:
```

Experiment: "checkout_test"
Salt: "abc123" (unique per experiment)
User: "user_456"

hash("user_456" + "checkout_test" + "abc123") % 100 = 37
37 < 50 → Variant A

Same user next time:
hash("user_456" + "checkout_test" + "abc123") % 100 = 37
Still Variant A (consistent!)

```


**For 70-30 Split**:
```

Assignment = hash(...) % 100

If Assignment < 70: Variant A (70%)
If Assignment >= 70: Variant B (30%)

```


**Advantages**:
- ✅ No database lookup needed (fast!)
- ✅ Works offline (deterministic algorithm)
- ✅ Scales infinitely (pure computation)
- ✅ No storage cost

**Disadvantages**:
- ❌ Can't manually reassign users
- ❌ Changing traffic allocation invalidates assignments

---

#### Approach 2: Database-Stored Assignment

**How It Works**:
```

1. User requests assignment
2. Check database for existing assignment
3. If exists → Return stored variant
4. If not → Calculate, store, return

Assignments Table:
user_id | experiment_id | variant_id | assigned_at
user_123 checkout_test variant_a 2025-01-08

```


**Advantages**:
- ✅ Can manually override assignments
- ✅ Can change traffic allocation (new users affected)
- ✅ Audit trail (know exactly who saw what)

**Disadvantages**:
- ❌ Database lookup required (slower)
- ❌ Storage grows with users × experiments
- ❌ Database bottleneck at scale

---

#### Recommended: Hybrid Approach

**Strategy**:
```

1. Use hash-based assignment (fast, scalable)

2. Cache result in Redis (even faster subsequent calls)
3. Store in database async (for audit, analysis)

First Request:

- Calculate hash (1ms)
- Store in Redis (1ms)
- Async store in database
- Return (2ms total)

Subsequent Requests:

- Check Redis (< 1ms)
- Return immediately

```
**This is how Optimizely and Google Optimize work!**

---

### 2. Statistical Significance Calculation

**The Core Question**: "Is Variant B really better, or just luck?"

**Metrics Needed**:
```

For each variant:

- Exposures (how many users saw it)
- Conversions (how many completed goal)
- Conversion rate = Conversions / Exposures

```
**Example**:
```

Variant A (Control):

- Exposures: 10,000
- Conversions: 1,000
- Conversion rate: 10%

Variant B (Treatment):

- Exposures: 10,000
- Conversions: 1,150
- Conversion rate: 11.5%

Question: Is 11.5% significantly better than 10%?
Or could this happen by chance?

```
**Statistical Tests**:

**Two-Sample T-Test**:
```

Used for: Continuous metrics (revenue, time spent)
Null hypothesis: Variant A and B have same mean
Calculate: t-statistic and p-value
Result: If p-value < 0.05, reject null (significant difference)

```
**Chi-Square Test**:
```

Used for: Conversion rates (binary outcomes)
Null hypothesis: Conversion rates are equal
Calculate: Chi-square statistic and p-value
Result: If p-value < 0.05, variants differ significantly

```
**Confidence Intervals**:
```

Variant A: 10% ± 0.6% (95% confidence)
→ True rate between 9.4% - 10.6%

Variant B: 11.5% ± 0.6% (95% confidence)
→ True rate between 10.9% - 12.1%

Since intervals don't overlap → Significant difference!

```
**Sample Size Calculator**:
```

To detect 10% relative lift with 95% confidence:

- Baseline conversion rate: 10%
- Minimum detectable effect: 1% absolute (10% relative)
- Power: 80%
- Required sample per variant: ~15,000 users

Run experiment until reach sample size

```


---

### 3. Targeting Rules

**Use Case**: Run experiment only for specific users

**Common Targeting Dimensions**:
```

Geographic:

- Country: US, UK, IN
- Region: California, Texas
- City: San Francisco

Demographics:

- Age: 18-24, 25-34
- Gender: Male, Female, Other

Platform:

- Device: Mobile, Desktop, Tablet
- OS: iOS, Android, Windows
- Browser: Chrome, Safari, Firefox

Behavioral:

- User segment: Premium, Free, Trial
- Previous purchases: Yes/No
- Account age: < 30 days, > 1 year

Custom:

- Company defined segments
- ML-based cohorts

```


**Example Targeting Rule**:
```

Experiment: "mobile_checkout_redesign"

Target:

- Platform: Mobile only
- Country: US, UK, Canada

- User segment: Active (purchased in last 30 days)

Exclusions:

- Internal employees
- Bots
- Users in other critical experiments

```
**Evaluation**:
```

User requests assignment:

1. Check if user matches ALL targeting criteria
2. If NO → User not in experiment (show default)
3. If YES → Assign to variant using hash

Benefits:

- Focus experiment on relevant users
- Avoid contamination
- Reduce noise in data

```
---

### 4. Traffic Allocation Strategies

#### Fixed Allocation
```

Example: 50-50 split

- 50% see Variant A
- 50% see Variant B
- Allocation stays constant

Use for: Traditional A/B tests

```
#### Gradual Rollout
```

Example: New feature launch

- Day 1: 5% see new feature
- Day 3: 10% see new feature

- Day 7: 25% see new feature
- Day 14: 50% see new feature
- Day 21: 100% see new feature

Use for: Risk mitigation, feature flags

```
#### Multi-Armed Bandit
```

Dynamic allocation based on performance:

- Start: 50-50 split
- After 1000 samples:
    - Variant A: 8% conversion
    - Variant B: 12% conversion
- Shift traffic: 30-70 split (favor B)
- Continuously optimize

Use for: Revenue optimization (show better variant to more users)

Trade-off: Slower to reach statistical significance

```
---

### 5. Real-Time Analytics Pipeline

**Architecture**:
```

[Event Tracking API]
↓
[Kafka Topics]
↓
[Stream Processor (Flink)]

- Aggregate by experiment/variant
- Calculate rates
- Sliding windows (hourly, daily)
  ↓
  [ClickHouse]
- Store aggregated metrics
- Fast queries for dashboards
  ↓
  [WebSocket Server]
- Push updates to dashboard

- Real-time visualization

```


**Aggregation Windows**:
```

1-minute window: For real-time monitoring
1-hour window: For hourly trends
1-day window: For daily summaries

Store all three for different use cases

```


**Why Stream Processing**:
- Real-time updates (< 1 minute delay)
- Handle billions of events
- Stateful aggregations
- Exactly-once semantics

---

### 6. Feature Flags Implementation

**Feature Flag vs A/B Test**:
```

A/B Test:

- Compare variants
- Statistical analysis
- Usually temporary (2-4 weeks)
- Goal: Find winner

Feature Flag:

- Gradual rollout
- Kill switch
- Can be permanent
- Goal: Safe deployment

```


**Rollout Strategy**:
```

Phase 1: Internal (0.1%)

- Test with employees
- Catch obvious bugs

Phase 2: Canary (1%)

- Real users, small sample
- Monitor errors/latency

Phase 3: Early Adopters (10%)

- Willing beta testers
- Gather feedback

Phase 4: General (50%)

- Half of users
- Validate at scale

Phase 5: Full Rollout (100%)

- Everyone gets new feature
- Monitor for issues

At each phase:

- Monitor error rates
- Check latency
- Track conversions
- If issues → Instant rollback

```

---

### 7. Assignment Caching Strategy

**Challenge**: 115K assignment QPS, can't hit database

**Solution: Multi-Layer Cache**

**L1 - Client-Side Cache** (Optional):
```

Store assignment in local storage/memory
TTL: Duration of session or experiment

Pros: Zero network calls
Cons: Can't change mid-session

```

**L2 - Redis Cache** (Primary):
```

Store all active assignments
Key: assign:{user_id}:{exp_id}
Value: {variant_id, config}
TTL: Experiment duration

Cache Stats:

- 99.9% hit ratio
- < 1ms latency
- Handles 115K QPS easily

```


**L3 - Database** (Source of Truth):
```

PostgreSQL stores all assignments
Used for:

- Analytics (who saw what)
- Audit trail
- Cache rebuilds

```


**Cache Warming**:
```

On experiment start:

- Pre-load experiment configs to Redis
- Pre-compute assignments for active users
- Ensures fast first request

```


---

## Scalability & Reliability

### Horizontal Scaling

**Assignment Service**:
```

Stateless service, scales linearly

- 115K QPS / 5K per instance = 23 instances
- Add buffer: 30 instances
- Auto-scale based on QPS

Cost: 30 × $100/month = $3,000/month

```
**Event Tracking Service**:
```

- 12K write QPS / 2K per instance = 6 instances
- Add buffer: 10 instances

Cost: 10 × $100/month = $1,000/month

```
**Kafka Cluster**:
```

- 3 brokers for high availability
- Replication factor: 3
- 10 partitions per topic (parallelism)

Cost: 3 × $500/month = $1,500/month

```
**Redis Cache**:
```

- Single instance sufficient (11 GB data)
- Or Redis Cluster for HA
- 3 nodes with replication

Cost: $200-500/month

```
**ClickHouse**:
```

- 5-node cluster
- Distributed queries
- Replicated for HA

Cost: 5 × $400/month = $2,000/month

```
---

### High Availability
```

```
**Critical Path - Assignment Service**:
```

Target: 99.99% availability

Strategies:

1. Deploy across 3 availability zones
2. Load balancer with health checks
3. Auto-scaling (replace failed instances)
4. Redis Cluster (no single point of failure)
5. Fallback: Return default variant if all fails

Max acceptable downtime: 52 minutes/year

```

**Non-Critical - Analytics**:
```

Target: 99.9% availability

Can tolerate:

- Analytics delay (process events later)
- Dashboard downtime (not on critical path)
- Report generation failures (retry)

```

---

### Data Consistency

**Experiment Configuration - Strong Consistency**:
```

Use PostgreSQL with ACID:

- Can't show wrong variant
- Changes must be atomic
- All servers see same config

Replication:

- Master-slave with read replicas
- Reads can have slight lag (acceptable)
- Writes always to master

```
```

```
**Event Data - Eventual Consistency**:
```

Use Kafka + Cassandra:

- OK if analytics delayed 1 minute
- Availability > consistency
- Can tolerate event loss (< 0.01%)

Why acceptable:

- Statistical analysis robust to small data loss
- Real-time dashboards approximate

```
---

### Disaster Recovery

**Backup Strategy**:
```

PostgreSQL (Experiment configs):

- Daily full backup
- Continuous WAL archiving
- Cross-region replication
- RPO: < 5 minutes

Kafka (Events):

- Replicated to 3 brokers
- Retention: 7 days
- Can replay events

ClickHouse (Analytics):

- Daily snapshots
- Rebuild from Kafka if needed
- RPO: < 1 hour (acceptable)

```
---

## Security & Privacy

### User Privacy
```

```
**PII Handling**:
```

DON'T Store:

- Names, emails, addresses
- Credit card numbers
- Personal information

DO Store:

- Hashed user IDs
- Anonymous identifiers
- Aggregated metrics only

```


**GDPR Compliance**:
```

User Rights:

- Right to deletion (remove all user data)
- Right to export (download experiment history)
- Right to opt-out (don't include in experiments)

Implementation:

- user_id hashing (can't reverse engineer)
- Data retention policies (auto-delete after 90 days)
- Opt-out flags in assignment service

```


---

### Experiment Security

**Access Control**:
```

Roles:

- Admin: Create, modify, delete experiments
- Analyst: View results, export data
- Developer: Read-only config access

Permissions:

- Experiment-level (can only modify your experiments)

- Company-level (can view all company experiments)

```
**Audit Logs**:
```

Track all changes:

- Who created experiment
- Who modified allocation
- Who stopped experiment early
- Who accessed results

Store in append-only log (tamper-proof)

```
---

## Performance Optimizations

### 1. Caching Experiment Configs

**Problem**: 10K experiments, 115K QPS, can't query DB each time

**Solution**:
```

Cache ALL experiment configs in Redis:

- 10K × 1 KB = 10 MB (tiny!)
- Update when experiment modified
- TTL: 5 minutes (or indefinite with active invalidation)

Assignment flow:

1. Check Redis for experiment config (< 1ms)
2. Calculate assignment (hash, < 1ms)
3. Return variant (< 2ms total)

vs. Database query:

1. Query PostgreSQL (5-10ms)
2. Calculate assignment (< 1ms)
3. Return (6-11ms total)

5x faster with cache!

```
---
```

```
### 2. Pre-Aggregation

**Problem**: Dashboard queries expensive (SUM over billions of events)

**Solution**:
```

Pre-aggregate metrics:

Raw Events (Kafka):
event1: user_123, checkout_test, variant_a, click
event2: user_456, checkout_test, variant_a, click
event3: user_789, checkout_test, variant_a, conversion

Stream Processing (Flink):
Aggregate every minute:

- checkout_test, variant_a: 1000 exposures, 50 conversions
- checkout_test, variant_b: 1000 exposures, 65 conversions

Store aggregates in ClickHouse:

- Query aggregates (fast!)
- Instead of scanning billions of raw events

Dashboard query:
SELECT SUM(exposures), SUM(conversions)
FROM aggregated_metrics
WHERE experiment_id = 'checkout_test'
AND date >= '2025-01-01'

Runs in < 100ms vs minutes for raw events

```

---

### 3. Handling Experiment Collisions

**Problem**: User in multiple experiments that affect same page

**Example**:
```

User assigned to:

- Experiment A: Test checkout button color (Blue vs Red)
- Experiment B: Test checkout layout (Single vs Multi-page)

Both experiments modify checkout page!
Risk: Results contaminated

```
**Solution 1: Mutual Exclusion**
```

Configuration:

- Experiment A: Exclude users in Experiment B
- Experiment B: Exclude users in Experiment A

Result:

- User in A → Not eligible for B
- User in B → Not eligible for A
- Clean experiment groups

Trade-off: Smaller sample size (users split between experiments)

```
**Solution 2: Orthogonal Experiments**
```

Design experiments to be independent:

- Experiment A: Homepage hero image
- Experiment B: Checkout button color
- Different pages, no interaction

Can run simultaneously without issues

```
**Solution 3: Layered Experiments**
```

Create experiment layers:

- Layer 1: Checkout experiments only
- Layer 2: Homepage experiments only
- Layer 3: Pricing experiments only

User can be in one experiment per layer

```
---

## Trade-offs & Alternatives
```

```
### 1. Hash-Based vs Stored Assignment

**Chose: Hash-Based (with cache)**

**Reasoning**:
- 115K QPS requires fast assignment
- Hash-based: < 1ms, scales infinitely
- Stored: 5-10ms, requires database scaling

**Trade-off**:
- Hash-based: Can't change allocation mid-experiment
- Stored: Flexible but slower

**Mitigation**:
- Use hash-based as default
- Cache in Redis for < 1ms
- Store async in database for analytics

---

### 2. Real-Time vs Batch Analytics

**Chose: Hybrid (Real-time + Batch)**

**Real-Time (Flink → ClickHouse)**:
```

For: Live dashboards
Latency: < 1 minute
Aggregation: Last 24 hours
Use case: Monitor running experiments

```


**Batch (Spark → Redshift)**:
```

For: Historical analysis
Latency: Hourly/daily
Aggregation: All time
Use case: Deep analysis, reports

```


**Why Both**:
- Real-time for monitoring
- Batch for accuracy and historical trends

---
```

```
### 3. ClickHouse vs Cassandra for Analytics

**Chose: ClickHouse**

**Reasoning**:
```

ClickHouse:
✅ Columnar (fast aggregations)
✅ Designed for analytics queries
✅ SQL interface (familiar)
✅ Real-time inserts + queries

Cassandra:
✅ Better for pure time-series writes
✅ Geographic distribution
❌ Aggregations slower
❌ No SQL (CQL different)

Decision: Analytics workload favors ClickHouse

```


---

### 4. Synchronous vs Asynchronous Event Tracking

**Chose: Asynchronous**

**Synchronous (Bad)**:
```

User clicks button:

1. Send tracking event (wait for response)
2. Button action proceeds

Problem: 100ms delay for user!

```


**Asynchronous (Good)**:
```

User clicks button:

1. Queue event locally (instant)
2. Button action proceeds immediately
3. Background: Batch send events

Result: No user-facing latency

```
---

## Technology Stack

| Layer | Technology | Purpose |
|-------|------------|---------|
| **CDN** | CloudFront | Dashboard static assets |
| **Load Balancer** | AWS ALB | Traffic distribution |
| **API Gateway** | Kong | Auth, rate limiting |
| **Assignment Service** | Go, Java | Fast variant assignment |
| **Event Service** | Node.js, Python | Event ingestion |
| **Analytics** | Apache Flink | Stream processing |
| **Cache** | Redis/Redis Cluster | Config & assignment cache |
| **Config DB** | PostgreSQL | Experiments, assignments |
| **Event Stream** | Apache Kafka | Durable event pipeline |
| **Analytics DB** | ClickHouse | Real-time queries |
| **Data Warehouse** | Amazon Redshift | Historical analysis |
| **Object Storage** | Amazon S3 | Raw event backup |
| **Monitoring** | Prometheus, Datadog | Metrics & alerts |

---

## Technology Decision Summary

### Complete Technology Stack with Justifications

| Component | Chosen | Why Chosen | Why NOT Alternative |
|-----------|--------|------------|---------------------|
| **Assignment Language** | **Go** | • 1-2ms latency<br>• Low memory (10-20MB)<br>• Native concurrency | ❌ Java: Higher memory<br>❌ Node.js: Variable latency<br>❌ Python: Too slow (5-10ms) |
| **Config Database** | **PostgreSQL** | • ACID transactions<br>• Complex JOINs<br>• JSON support<br>• Relationships | ❌ MongoDB: No transactions<br>❌ DynamoDB: No complex queries<br>❌ MySQL: Weaker JSON support |
| **Cache** | **Redis** | • Sub-ms latency<br>• Rich data structures<br>• Pub/Sub for invalidation<br>• Cluster mode | ❌ Memcached: No data structures<br>❌ Hazelcast: Slower, heavier<br>❌ In-memory: No shared state |
| **Event Stream** | **Kafka** | • High throughput<br>• Replay capability<br>• Multiple consumers<br>• Durability | ❌ RabbitMQ: Lower throughput, no replay<br>❌ SQS: Expensive, no replay<br>❌ Pub/Sub: Vendor lock-in |
| **Stream Processing** | **Flink** | • True streaming (ms)<br>• Event-time windows<br>• Stateful processing<br>• Exactly-once | ❌ Spark: Micro-batch (seconds)<br>❌ Kafka Streams: Less features<br>❌ Storm: Older, less mature |
| **Analytics DB** | **ClickHouse** | • Columnar (fast aggregations)<br>• 10-50x compression<br>• Real-time queries<br>• SQL | ❌ Cassandra: Slow aggregations<br>❌ PostgreSQL: Doesn't scale<br>❌ BigQuery:
```

```
Expensive, vendor lock-in |
| **Data Warehouse** | **Redshift** | • SQL interface<br>• Integrates
with S3<br>• Cost-effective<br>• Mature | ❌ Snowflake: More
expensive<br>❌ BigQuery: Vendor lock-in<br>❌ Direct S3 queries: Too
slow |

### Critical Path Analysis
```

Assignment Request (< 10ms requirement):

┌──┐

| Component Time Why This Tech |

├──┤

| API Gateway 0.5ms Kong (lightweight) |

| Assignment Service 0.5ms Go (fast concurrency) |

| Redis GET 1ms In-memory, sub-ms access |

| Hash Calculation 0.5ms MD5, O(1) operation |

| Network 2ms Within same AZ |

| Total ~4.5ms ✅ Well under 10ms budget |

└──┘

If PostgreSQL (no cache):

| Database Query 5-10ms Too slow! |

| Total ~8-12ms ❌ Risk exceeding budget |

```
### Event Processing Path
```

Event Tracking (< 100ms requirement):

┌──┐

| Component Time Why This Tech |

├──┤

| SDK (async) ~0ms Non-blocking |

| API Gateway 0.5ms Auth + validation |

| Event Service 1ms Node.js (I/O optimized) |

| Kafka Write 2-5ms Disk-backed, replicated |

| Response 0.5ms 200 OK returned |

| Total ~4-7ms ✅ Client sees <10ms |

| |

| Async Processing: |

| Flink Processing <1sec Real-time aggregation |

| ClickHouse Write <1sec Batch insert |

| Dashboard Update <1min WebSocket push |

└──┘

```
### Why Specific Tech Combinations Work Together

**1. Kafka + Flink + ClickHouse (Analytics Pipeline)**
```

Why this combination is perfect:

Kafka:

- Produces ordered, durable event stream
- Multiple consumers can read same data
- Replay for reprocessing

Flink:

- Consumes from Kafka in real-time
- Stateful windowed aggregations
- Exactly-once processing

ClickHouse:

- Stores Flink's aggregated results
- Fast queries on aggregated data
- Columnar storage perfect for metrics

Result: Real-time analytics pipeline with <1 minute latency

Alternative combinations considered:
❌ Kafka + Spark + Cassandra: Spark has seconds delay
❌ SQS + Lambda + DynamoDB: No replay, harder to scale
❌ Pub/Sub + Dataflow + BigQuery: Vendor lock-in, expensive

```
**2. Go + Redis + PostgreSQL (Assignment Service)**
```

Why this combination is perfect:

Go:

- Fast hash calculation (1-2ms)
- Low memory footprint
- Handles 115K QPS per instance

Redis:

- Caches experiment configs (10 MB total)

- 99.9% hit ratio (sub-ms lookups)
- Handles misses gracefully

PostgreSQL:

- Async storage for assignments
- Audit trail for compliance
- Source of truth for rebuilds

Result: <10ms p99 latency achieved

Without Redis:
❌ 115K QPS would need 575 database connections
❌ PostgreSQL can't handle that load
❌ Would need expensive database scaling

```
### Cost-Performance Trade-off Analysis
```

Option 1: All Managed Services (Easy but Expensive)
├── DynamoDB instead of PostgreSQL: +$2,000/mo
├── Kinesis instead of Kafka: +$3,000/mo
├── BigQuery instead of ClickHouse: +$10,000/mo
├── Managed Redis (ElastiCache): +$500/mo
└── Total: ~$36,000/mo (75% more expensive)

Pros: Less operational overhead
Cons: 75% higher cost, vendor lock-in

Option 2: Self-Hosted (Chosen - Best Balance)
├── Self-hosted Kafka: $1,500/mo
├── Self-hosted ClickHouse: $2,000/mo
├── RDS PostgreSQL: $300/mo
├── ElastiCache Redis: $450/mo
└── Total: ~$20,500/mo

Pros: Cost-effective, no vendor lock-in, full control
Cons: More operational overhead (but manageable)

Option 3: Minimize Costs (Limited Scale)
├── Single PostgreSQL for everything: $500/mo
├── No Kafka (direct DB writes): $0
├── No Redis (use PostgreSQL): $0
└── Total: ~$500/mo

Pros: Very cheap
Cons: Can't scale beyond 1K QPS, slow queries

Use when: MVP, small scale only

```
---

## Interview Talking Points

### Key Design Decisions

1. **Why hash-based assignment?**
   - 115K QPS requires < 10ms latency
   - Deterministic without database lookup
   - Scales infinitely

2. **Why Redis cache?**
   - All configs fit in memory (10 MB)
   - 99.9% cache hit ratio
   - < 1ms latency

3. **Why Kafka?**
   - Buffer traffic spikes
   - Multiple consumers (analytics, warehouse)
   - Can replay events for debugging

4. **Why ClickHouse?**
   - Columnar storage perfect for aggregations
   - Real-time queries on billions of rows
   - Better than Cassandra for analytics

5. **Why PostgreSQL for configs?**
   - ACID transactions (critical)
   - Small dataset (10K experiments)
   - Complex queries needed

6. **Why stream processing?**
   - Real-time dashboards (< 1 minute delay)
   - Stateful aggregations
   - Handle billions of events

### Potential Bottlenecks & Solutions

1. **Assignment service overwhelmed**
   - Solution: Cache configs, hash-based assignment, horizontal scaling

2. **Event ingestion slow**
   - Solution: Kafka buffering, batch processing, async

3. **Analytics queries slow**
   - Solution: Pre-aggregation, columnar storage (ClickHouse)

4. **Cache invalidation**
   - Solution: TTL + active invalidation, Redis Cluster
```

```
5. **Database writes**
   - Solution: Async writes, batch inserts, Kafka buffer

---

## Real-World Architectures

### Optimizely's Architecture
```

Assignment: Edge network (< 10ms globally)
Events: Sent to nearest datacenter
Analytics: Stream processing + data warehouse
Scale: 12 billion decisions/day

```


### Google Optimize
```

Assignment: Google's global CDN
Events: Google Analytics integration
Analytics: BigQuery
Scale: Integrated with Google ecosystem

```


### LaunchDarkly
```

Assignment: In-memory SDKs (offline-first)
Events: Event streaming pipeline
Analytics: Real-time + historical
Scale: Feature flags at massive scale

```


---

## Advanced Features

### 1. Holdout Groups
```

Purpose: Control group across multiple experiments

Example:

- 90% of users: Participate in various experiments
- 10% holdout: See original experience (no experiments)

Benefits:

- Measure overall impact of experimentation program
- Detect interaction effects
- Validate that experiments actually improve metrics

```
### 2. Sequential Testing
```

Purpose: Stop experiment early when clear winner

Traditional: Wait for full sample size (2-4 weeks)
Sequential: Check significance continuously, stop early if clear winner

Benefits:

- Faster decisions (days vs weeks)
- Reduce risk (stop bad experiments sooner)
- Opportunity cost (ship winners faster)

Trade-off:

- More complex statistics
- Risk of false positives (solved with adjusted thresholds)

```
### 3. Revenue Attribution
```

Purpose: Calculate revenue impact of experiments

Track:

- Direct revenue (during experiment)
- Lifetime value (long-term impact)
- Per-user revenue

Example:
Variant A: $10 revenue per user
Variant B: $12 revenue per user
With 10M users: $20M additional revenue/year

Business value: Clear ROI for winning variant

```
---

## Detailed Component Interactions
```

```
### Complete Assignment Flow
```

1. User visits website
2. Website SDK calls: POST /api/v1/assign
3. API Gateway: Authenticates request
4. Assignment Service:
   a. Check Redis for cached assignment (< 1ms)
   b. If miss: Calculate hash assignment (< 1ms)
   c. Check targeting rules (user matches?)
   d. Cache in Redis
   e. Async store in PostgreSQL
5. Return variant to website (< 10ms total)
6. Website renders appropriate variant

```


### Complete Event Tracking Flow
```

1. User completes action (purchase, signup, etc.)
2. Website SDK: POST /api/v1/track (async, doesn't block)
3. Event Service:
   a. Validates event
   b. Enriches with metadata
   c. Writes to Kafka topic
4. Returns 200 OK (< 100ms)
5. Kafka consumers process:
   a. Flink: Real-time aggregation
   b. Spark: Batch processing
   c. Warehouse: Historical storage
6. Dashboards update automatically via WebSocket

```


---

## Cost Analysis

### Monthly Infrastructure Costs

**For 100M users, 1B events/day, 10K experiments**:

```

Assignment Service: 30 instances × $100 = $3,000
Event Service: 10 instances × $100 = $1,000
Analytics Service: 5 instances × $200 = $1,000

Load Balancers: 2 × \$25 = \$50

PostgreSQL (RDS): db.r5.large = \$300
Redis Cluster: 3 nodes × \$150 = \$450

Kafka: 3 brokers × \$500 = \$1,500
ClickHouse: 5 nodes × \$400 = \$2,000
Redshift: \$1,000

Flink/Spark: \$1,500
S3 Storage (365 TB): \$8,000

Monitoring: \$500
CDN: \$200

Total: ~\$20,500/month

Revenue Model:

- Price per experiment: \$500/month
- 1000 paying customers = \$500K/month revenue
- Margin: 96% (very profitable!)

```


---

## References & Further Reading

### Statistical Resources
1. **"Trustworthy Online Controlled Experiments"** - Microsoft Research
2. **"A/B Testing: The Most Powerful Way to Turn Clicks Into
Customers"** - Dan Siroker
3. **"Statistical Methods in Online A/B Testing"** - Google Research

### System Design Resources
1. **Optimizely Engineering Blog**
2. **Netflix Experimentation Platform** - Tech blog
3. **Booking.com A/B Testing**- Engineering blog

### Tools & Platforms
1. **Optimizely**: Industry leader
2. **Google Optimize**: Free tier available
3. **LaunchDarkly**: Feature flags focused
4. **Split.io**: Enterprise platform
5. **VWO**: Visual editor included

---

## Appendix


```

```
### Sample Size Calculations
```

Given:

- Baseline conversion rate: 5%
- Minimum detectable effect: 10% relative (0.5% absolute)
- Statistical power: 80%
- Significance level: 95%

Required sample per variant: ~25,000 users

Experiment duration:

- Daily visitors: 10,000
- Days needed: 5 days (2.5 days per variant)

General formula:
$n = 16 \times p \times (1-p) / (MDE)^2$
Where:

- p = baseline rate
- MDE = minimum detectable effect

```
### Statistical Significance Thresholds
```

P-value	Confidence	Interpretation
< 0.001	99.9%	Extremely significant
< 0.01	99%	Highly significant
< 0.05	95%	Significant (standard)
< 0.10	90%	Marginally significant

> 0.10 < 90% Not significant

Industry standard: $p < 0.05$ (95% confidence)
Conservative: $p < 0.01$ (99% confidence)

```
### Common Experiment Durations
```

Traffic Level	Sample Size	Duration
Low (1K/day)	10K	10 days
Medium (10K/day)	10K	1 day

High (100K/day) 10K 3 hours
Very High (1M/day) 10K 15 minutes

Rule of thumb: Run for at least 1 week to capture:

- Weekday vs weekend behavior
- Time-of-day patterns
- Seasonality

```

---

**Document Version**: 1.0
**Last Updated**: January 8, 2025
**Author**: System Design Interview Prep
**Status**: Complete & Interview-Ready ✅

**Pro Tip**: A/B testing platforms are common interview questions at
companies like Optimizely, Google, Facebook, Netflix, and any data-
driven company. Master this design!
```