Logging and Metrics System Design - High-Level Design Focus

Table of Contents

Problem Statement & Requirements

Problem Definition

Design a distributed observability system that can:

- Collect and aggregate logs from 10,000+ servers
- Store and query metrics with millisecond precision
- Process 1TB+ of logs per day
- Support real-time alerting and dashboards
- Retain data for compliance (90 days hot, 1 year cold)

Use Cases

1. Logging System (ELK-style)

- Application error tracking and debugging
- Security audit logs
- User activity tracking
- System performance logs
- Business analytics

2. Metrics System (Prometheus-style)

- Real-time monitoring (CPU, memory, disk)
- Application performance (latency, throughput, error rate)
- Business metrics (orders/sec, revenue, user signups)
- Custom metrics from applications

- Infrastructure health

Functional Requirements

Logging

1. **Collection**: Ingest logs from multiple sources (apps, systems, containers)
2. **Parsing**: Extract structured data from unstructured logs
3. **Search**: Full-text search with millisecond latency
4. **Aggregation**: Group and aggregate log data
5. **Alerting**: Trigger alerts on log patterns

Metrics

1. **Collection**: Scrape metrics from endpoints at configurable intervals
2. **Storage**: Efficient time-series storage
3. **Querying**: Query language for analysis and aggregation
4. **Visualization**: Real-time dashboards and graphs
5. **Alerting**: Alert on metric thresholds and anomalies

Non-Functional Requirements

1. **Performance**
    - Log ingestion: 1M events/second
    - Metrics ingestion: 10M samples/second
    - Query latency: <1 second for recent data
    - Dashboard refresh: <5 seconds
2. **Scalability**
    - Handle 10,000+ hosts
    - Store 1TB+ logs/day
    - Store 100M+ time-series
    - Support 1,000+ concurrent users
3. **Availability**
    - 99.9% uptime for ingestion
    - 99.95% uptime for queries
    - No data loss during failures
    - Graceful degradation
4. **Retention**
    - Hot storage: 7-90 days (fast queries)
    - Cold storage: 1-7 years (slower, cheaper)
    - Configurable per data source

Capacity Estimation

```
Logging System:
- Servers: 10,000
- Logs per server: 1,000 lines/minute
- Total log rate: 10M lines/minute = 167K lines/second
- Average log size: 500 bytes
- Ingestion rate: 83 MB/second = 7.2 TB/day
- Storage (90 days): 648 TB

Metrics System:
- Servers: 10,000
- Metrics per server: 100 metrics
- Scrape interval: 15 seconds
- Total metrics: 1M metrics
- Samples per day: 1M × (86400/15) = 5.76B samples
- Sample size: 16 bytes (timestamp + value)
- Daily storage: 92 GB
- Storage (90 days): 8.3 TB

Total Storage: ~656 TB (hot) + archives (cold)

Infrastructure:
- Ingestion nodes: 20-30 (handle peaks)
- Storage nodes: 50-100 (distributed)
- Query nodes: 10-20 (dedicated)
- Cache nodes: 5-10 (Redis for hot data)

Cost Estimate (AWS):
- Compute (EC2): $15,000/month
- Storage (EBS/S3): $25,000/month
- Network: $5,000/month
- Total: ~$45,000/month
```

High-Level Architecture

System Overview

```
┌─────────────────────────────────────────────────────────────────┐
│                        DATA SOURCES                              │
├─────────────────────────────────────────────────────────────────┤
│  [Applications] [Containers] [VMs] [Cloud Services] [Network]   │
│       ↓              ↓           ↓          ↓              ↓     │
│  [Filebeat]   [Docker logs] [Syslog] [CloudWatch]  [SNMP]      │
└─────────────────────────────────────────────────────────────────┘
                             │
                             │
                             ▼
┌─────────────────────────────────────────────────────────────────┐
│                    INGESTION LAYER                               │
├─────────────────────────────────────────────────────────────────┤
```

```
│                                                      │
│  ┌──────────────────┐      ┌──────────────────┐      │
│  │ Logs Pipeline    │      │ Metrics Pipeline │      │
│  │                  │      │                  │      │
│  │ ┌──────────────┐ │      │ ┌──────────────┐ │      │
│  │ │  Logstash    │ │      │ │  Prometheus  │ │      │
│  │ │  (Parser &   │ │      │ │  (Scraper)   │ │      │
│  │ │  Transform)  │ │      │ │              │ │      │
│  │ └──────┬───────┘ │      │ └──────┬───────┘ │      │
│  │        │         │      │        │         │      │
│  │ ┌──────▼───────┐ │      │ ┌──────▼───────┐ │      │
│  │ │Message Queue │ │      │ │Message Queue │ │      │
│  │ │   (Kafka)    │ │      │ │   (Kafka)    │ │      │
│  │ └──────────────┘ │      │ └──────────────┘ │      │
│  └──────────────────┘      └──────────────────┘      │
└─────────────────────────┬────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│                   STORAGE LAYER                      │
├──────────────────────────────────────────────────────┤
│                                                      │
│  ┌──────────────────┐      ┌──────────────────┐      │
│  │ Log Storage      │      │ Metrics Storage  │      │
│  │                  │      │                  │      │
│  │ ┌──────────────┐ │      │ ┌──────────────┐ │      │
│  │ │Elasticsearch │ │      │ │  Prometheus  │ │      │
│  │ │  Cluster     │ │      │ │    TSDB      │ │      │
│  │ │ (Hot Data)   │ │      │ │ (Hot Data)   │ │      │
│  │ └──────┬───────┘ │      │ └──────┬───────┘ │      │
│  │        │         │      │        │         │      │
│  │ ┌──────▼───────┐ │      │ ┌──────▼───────┐ │      │
│  │ │  S3/Glacier  │ │      │ │  Thanos/     │ │      │
│  │ │ (Cold Data)  │ │      │ │  Cortex      │ │      │
│  │ └──────────────┘ │      │ │ (Long-term)  │ │      │
│  └──────────────────┘      │ └──────────────┘ │      │
│                            └──────────────────┘      │
└─────────────────────────┬────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
│                    QUERY LAYER                       │
├──────────────────────────────────────────────────────┤
│                                                      │
│  ┌────────────┐   ┌────────────┐   ┌────────────┐    │
│  │  Kibana    │   │  Grafana   │   │ API Layer  │    │
│  │(Log Search)│   │(Dashboards)│   │(Custom Apps)│   │
│  └────────────┘   └────────────┘   └────────────┘    │
│                                                      │
└─────────────────────────┬────────────────────────────┘
                          │
                          ▼
┌──────────────────────────────────────────────────────┐
```

```
│                    ALERTING LAYER                                  │
├────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  ┌────────────────────────────────────────────────────────────┐   │
│  │          Alert Manager & Notification Service              │   │
│  │                                                             │   │
│  │  ┌─────────┐  ┌──────────┐  ┌────────┐  ┌──────────┐    │   │
│  │  │PagerDuty│  │  Slack   │  │  Email │  │  Webhook │    │   │
│  │  └─────────┘  └──────────┘  └────────┘  └──────────┘    │   │
│  └────────────────────────────────────────────────────────────┘   │
└────────────────────────────────────────────────────────────────────┘
```

Architectural Patterns

1. Lambda Architecture (Batch + Stream)

```
Real-time Layer (Stream):
├── Kafka → Stream Processor → Hot Storage
├── Latency: <1 second
├── Retention: 7-90 days
└── Use: Dashboards, real-time alerts

Batch Layer:
├── S3 → Spark Jobs → Data Warehouse
├── Latency: Hours
├── Retention: Years
└── Use: Analytics, compliance, ML training

Serving Layer:
├── Combines results from both
├── Provides unified query interface
└── Handles data quality and corrections
```

2. Kappa Architecture (Stream Only)

```
Single Pipeline:
├── Kafka → Stream Processor → Storage
├── Reprocess historical data if needed
├── Simpler than Lambda
└── Preferred for most use cases

Benefits:
✓ Single codebase
✓ Easier operations
✓ Sufficient for most requirements
```

Key Design Decisions

1. Push vs Pull Model

Logs: Push Model (Recommended)

```
Why:
- Logs are generated continuously
- Applications can't be "polled"
- Need immediate delivery for errors
- Buffering handles backpressure

Implementation:
App → Agent (Filebeat) → Logstash → Kafka → Storage
```

Metrics: Pull Model (Prometheus)

```
Why:
- Service discovery easier
- Target health visible
- Prevents overwhelming targets
- Metrics collected at consistent intervals

Implementation:
Prometheus → Scrape → App /metrics endpoint
```

Metrics: Push Model (Alternative)

```
Why:
- Short-lived jobs (cron, Lambda)
- Behind firewalls
- Dynamic infrastructure

Implementation:
App → Push Gateway → Prometheus
```

2. Hot vs Cold Storage Tiering

```
Hot Tier (Fast, Expensive):
├── Recent data (7-90 days)
├── SSD storage
├── High IOPS
├── Low latency queries (<100ms)
└── Examples: Elasticsearch, Prometheus
```

```
Warm Tier (Medium):
├── Less recent (90-180 days)
├── Slower SSD or fast HDD
├── Medium IOPS
├── Acceptable latency (1-5s)
└── Examples: S3 Standard, Clickhouse

Cold Tier (Slow, Cheap):
├── Archives (>180 days)
├── Object storage
├── Low IOPS
├── Higher latency (minutes)
└── Examples: S3 Glacier, Google Coldline

Benefits:
✓ 80% cost reduction
✓ Meets compliance requirements
✓ Transparent to users (federated queries)
```

3. Centralized vs Distributed

```
Centralized Collection:
All logs → Central cluster

Pros:
✓ Simple operations
✓ Easier to query
✓ Lower complexity

Cons:
✗ Single point of failure
✗ Network bandwidth
✗ Compliance issues (data locality)

Distributed (Recommended):
Regional clusters → Federated queries

Pros:
✓ Data locality
✓ Reduced network costs
✓ Failure isolation
✓ Compliance friendly

Cons:
✗ Complex operations
✗ Harder to query across regions
```

Core Components

1. Log Collection Agents

Purpose: Collect and ship logs from sources

Filebeat (Elastic Stack)

```
Features:
├── Lightweight (Go-based)
├── Tails log files
├── Handles log rotation
├── Built-in modules (nginx, mysql, etc.)
├── Multiline support
└── Back-pressure handling

Configuration:
filebeat.inputs:
  - type: log
    paths:
      - /var/log/app/*.log
    fields:
      app: my-app
      env: production

output.logstash:
  hosts: ["logstash-1:5044", "logstash-2:5044"]
  loadbalance: true
```

Fluentd (CNCF)

```
Features:
├── Plugin ecosystem (500+ plugins)
├── JSON structured logging
├── Memory and file buffering
├── Tag-based routing
└── Kubernetes native

Configuration:
<source>
  @type tail
  path /var/log/app/*.log
  tag app.logs
  <parse>
    @type json
  </parse>
</source>
```

```
<match app.logs>
  @type kafka2
  brokers kafka1:9092,kafka2:9092
  topic application-logs
</match>
```

2. Log Processing Pipeline

Purpose: Parse, transform, enrich logs

Logstash

```
Pipeline Stages:

Input → Filter → Output

Input:
├── Beats (Filebeat, Metricbeat)
├── Kafka
├── HTTP
├── S3
└── Database CDC

Filter:
├── Grok (pattern matching)
├── JSON parser
├── Date parser
├── GeoIP enrichment
├── User-Agent parsing
└── Custom Ruby code

Output:
├── Elasticsearch
├── Kafka
├── S3
├── Multiple outputs
```

Example Pipeline:

```
input {
  beats {
    port => 5044
  }
}

filter {
  # Parse nginx access logs
  grok {
```

```
    match => {
      "message" => '%{IPORHOST:clientip} - - \[%{HTTPDATE:timestamp}\] "%{WORD:method} %{URIPATHPARAM:request} HTTP/%{NUMBER:httpversion}" %{NUMBER:response} %{NUMBER:bytes}'
    }
  }

  # Convert response to integer
  mutate {
    convert => { "response" => "integer" }
    convert => { "bytes" => "integer" }
  }

  # Add GeoIP data
  geoip {
    source => "clientip"
  }

  # Parse timestamp
  date {
    match => [ "timestamp", "dd/MMM/YYYY:HH:mm:ss Z" ]
    target => "@timestamp"
  }
}

output {
  elasticsearch {
    hosts => ["es-1:9200", "es-2:9200"]
    index => "nginx-logs-%{+YYYY.MM.dd}"
  }
}
```

3. Message Queue (Kafka)

Purpose: Buffer and decouple ingestion from storage

```
Why Kafka?
├── High throughput (millions/sec)
├── Durability (replication)
├── Ordered delivery
├── Replay capability
├── Backpressure handling
└── Multiple consumers

Topic Design:
├── application-logs (partitioned by app_id)
├── system-metrics (partitioned by host)
├── error-logs (single partition for ordering)
└── audit-logs (compacted for deduplication)
```

```
Configuration:
├── Replication factor: 3
├── Retention: 7 days
├── Compression: LZ4
└── Batch size: 1MB
```

4. Log Storage (Elasticsearch)

Purpose: Store and index logs for fast search

```
Architecture:
├── Master nodes (3): Cluster coordination
├── Data nodes (20+): Store and query data
├── Ingest nodes (5): Pre-processing
├── Coordinating nodes (5): Route requests
└── Machine learning nodes (3): Anomaly detection

Index Structure:
logs-application-2025.01.08
├── Shards: 5 primary
├── Replicas: 1 per shard
├── Refresh interval: 5s (near real-time)
└── Lifecycle: Hot → Warm → Cold → Delete

Index Template:
{
  "index_patterns": ["logs-*"],
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1,
    "index.lifecycle.name": "logs-policy"
  },
  "mappings": {
    "properties": {
      "@timestamp": {"type": "date"},
      "level": {"type": "keyword"},
      "message": {"type": "text"},
      "host": {"type": "keyword"},
      "app_id": {"type": "keyword"}
    }
  }
}
```

Index Lifecycle Management:

```
Phase 1: Hot (0-7 days)
├── Fast SSDs
├── High resource allocation
```

```
├── All queries hit this tier
└── Rollover at 50GB or 7 days

Phase 2: Warm (7-30 days)
├── Slower SSDs
├── Reduced replicas (1 → 0)
├── Force merge to reduce segments
└── Shrink shards (5 → 1)

Phase 3: Cold (30-90 days)
├── Object storage (S3)
├── Searchable snapshots
├── Slower queries (acceptable)
└── 90% cost reduction

Phase 4: Delete (>90 days)
├── Remove from cluster
├── Optional archive to Glacier
└── Meets retention policy
```

5. Metrics Collection (Prometheus)

Purpose: Scrape and store time-series metrics

```
Architecture:
Prometheus Server
├── Scraper (pulls metrics)
├── TSDB (time-series database)
├── Query engine (PromQL)
└── Alertmanager integration

Service Discovery:
├── Static configs
├── Kubernetes (pods, services)
├── AWS (EC2, ECS)
├── Consul, etcd
└── DNS-based

Scrape Configuration:
scrape_configs:
  - job_name: 'api-servers'
    scrape_interval: 15s
    kubernetes_sd_configs:
      - role: pod
        namespaces:
          names: ['production']
    relabel_configs:
      - source_labels: [__meta_kubernetes_pod_label_app]
        action: keep
        regex: api-server
```

Metrics Types:

```
Counter (always increasing):
  http_requests_total 12345

Gauge (can go up/down):
  memory_usage_bytes 8589934592

Histogram (distribution):
  http_request_duration_seconds_bucket{le="0.1"} 100
  http_request_duration_seconds_bucket{le="0.5"} 450
  http_request_duration_seconds_bucket{le="1.0"} 900

Summary (percentiles):
  http_request_duration_seconds{quantile="0.5"} 0.23
  http_request_duration_seconds{quantile="0.9"} 0.87
  http_request_duration_seconds{quantile="0.99"} 1.23
```

6. Time-Series Database (Prometheus TSDB)

Purpose: Efficient storage for metrics

```
Storage Format:
├── Chunks: 2-hour blocks of compressed data
├── Indexes: Fast lookup by labels
├── WAL: Write-ahead log for durability
└── Compaction: Merge old chunks

Storage Efficiency:
├── Compression: ~1.5 bytes per sample
├── Sample: 16 bytes → Compressed: 1.5 bytes
├── 1M metrics × 15s interval = 240B samples/day
├── Storage: 240B × 1.5 bytes = 360 GB/day
└── Actual with overhead: ~400 GB/day

Retention:
├── Local: 15 days (fast queries)
├── Remote: Thanos/Cortex (long-term)
└── Downsampling: 5m → 1h → 6h aggregates
```

7. Query Engines

Elasticsearch Query DSL

```
{
  "query": {
    "bool": {
      "must": [
        {"term": {"level": "ERROR"}},
        {"range": {"@timestamp": {
          "gte": "now-1h"
        }}}
      ],
      "filter": [
        {"term": {"app_id": "payment-service"}}
      ]
    }
  },
  "aggs": {
    "errors_over_time": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "5m"
      }
    }
  },
  "size": 100
}
```

PromQL (Prometheus Query Language)

```
# Request rate (requests/second)
rate(http_requests_total[5m])

# 99th percentile latency
histogram_quantile(0.99,
  rate(http_request_duration_seconds_bucket[5m])
)

# Error rate percentage
sum(rate(http_requests_total{status="500"}[5m]))
/
sum(rate(http_requests_total[5m])) * 100

# Memory usage by pod
container_memory_usage_bytes{pod=~"api-.*"}

# Alert condition
avg(cpu_usage_percent) > 80
```

8. Visualization Layer

Kibana (Elastic Stack)

```
Features:
├── Discover: Ad-hoc log exploration
├── Visualize: Charts, graphs, maps
├── Dashboard: Combine visualizations
├── Canvas: Pixel-perfect presentations
├── Machine Learning: Anomaly detection
└── Alerting: Built-in alert rules

Common Use Cases:
├── Application error tracking
├── Security event analysis
├── Business analytics
├── User behavior analysis
└── Compliance reporting
```

Grafana (Metrics)

```
Features:
├── Time-series visualizations
├── Multi-datasource support
├── Template variables
├── Alert rules and notifications
├── Dashboard sharing
└── Plugin ecosystem

Common Dashboards:
├── System metrics (CPU, memory, disk)
├── Application performance (RED metrics)
├── Business metrics (orders, revenue)
├── SLO/SLI tracking
└── Cost monitoring
```

9. Alerting System

Purpose: Notify on-call team of issues

```
Alert Manager (Prometheus)
├── Alert routing
├── Grouping & deduplication
├── Silencing
├── Inhibition rules
└── Notification channels

Alert Definition:
```

```
groups:
  - name: api_alerts
    rules:
      - alert: HighErrorRate
        expr: |
          sum(rate(http_requests_total{status="500"}[5m]))
          /
          sum(rate(http_requests_total[5m]))
          > 0.01
        for: 5m
        labels:
          severity: critical
          team: backend
        annotations:
          summary: "High error rate on {{ $labels.instance }}"
          description: "Error rate is {{ $value }}%"

Notification Routing:
route:
  receiver: 'default'
  group_by: ['alertname', 'cluster']
  routes:
    - match:
        severity: critical
      receiver: 'pagerduty'
    - match:
        severity: warning
      receiver: 'slack'

Receivers:
- name: 'pagerduty'
  pagerduty_configs:
    - service_key: '<key>'
- name: 'slack'
  slack_configs:
    - channel: '#alerts'
```

Storage Architecture

Log Storage Optimization

1. Index Partitioning

```
Time-Based Sharding:
logs-app-2025.01.08
├── Contains: Jan 8, 2025 data
├── Benefit: Easy to delete old data
├── Benefit: Queries often time-bound
└── Rollover: Daily or at 50GB
```

```
Service-Based Sharding:
logs-payment-2025.01
logs-user-2025.01
logs-notification-2025.01
├── Benefit: Isolate noisy services
├── Benefit: Different retention policies
└── Trade-off: More indexes to manage

Hybrid Approach (Recommended):
logs-{service}-{date}
├── Best of both worlds
├── Example: logs-payment-2025.01.08
└── Scales with both dimensions
```

2. Index Templates & Mappings

```
{
  "index_patterns": ["logs-*"],
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1,
    "refresh_interval": "5s",
    "index": {
      "codec": "best_compression",
      "sorting": {
        "fields": ["@timestamp"]
      }
    }
  },
  "mappings": {
    "properties": {
      "@timestamp": {
        "type": "date",
        "format": "strict_date_optional_time||epoch_millis"
      },
      "level": {
        "type": "keyword"
      },
      "message": {
        "type": "text",
        "fields": {
          "keyword": {
            "type": "keyword",
            "ignore_above": 256
          }
        }
      },
      "host": {
        "type": "keyword"
```

```
      },
      "trace_id": {
        "type": "keyword"
      },
      "numeric_fields": {
        "type": "integer"
      }
    }
  }
}
```

Key Decisions:

- `text` for full-text search (tokenized)
- `keyword` for exact matching, aggregations
- `date` for time-based queries
- Compression for storage savings

3. Shard Allocation

```
Shard Sizing:
├── Target: 30-50 GB per shard
├── Too small: Management overhead
├── Too large: Slow recovery, rebalancing
└── Daily index: Adjust shards based on volume

Formula:
shards = ceiling(daily_volume_GB / 40)

Example:
├── Daily volume: 200 GB
├── Shards: 200 / 40 = 5 shards
└── Per shard: ~40 GB

Replicas:
├── Production: 1 replica (2 copies total)
├── Critical: 2 replicas (3 copies)
├── Dev: 0 replicas (cost optimization)
```

Metrics Storage Optimization

1. Label Strategy

```
Good Labels (low cardinality):
├── job: "api-server"
├── instance: "10.0.1.5:9090"
├── region: "us-west-2"
```

```
├── env: "production"
└── Total combinations: <10,000

Bad Labels (high cardinality):
✗ user_id: "user123456"
✗ request_id: "uuid-1234-5678"
✗ email: "user@example.com"
└── Causes: Memory explosion, slow queries

Rule of Thumb:
- Labels should have <100 unique values
- Label combinations <10,000 per metric
- Use labels for dimensions you'll query by
```

2. Downsampling

```
Raw Data (15s interval):
├── Resolution: 15 seconds
├── Retention: 15 days
├── Storage: 100 GB
└── Use: Recent detailed analysis

5-Minute Aggregates:
├── Resolution: 5 minutes (20x reduction)
├── Retention: 90 days
├── Storage: 15 GB
└── Use: Weekly/monthly trends

1-Hour Aggregates:
├── Resolution: 1 hour (240x reduction)
├── Retention: 1 year
├── Storage: 5 GB
└── Use: Yearly trends, capacity planning

Thanos Configuration:
downsample:
  - from: 0s
    to: 40h
    resolution: raw
  - from: 40h
    to: 90d
    resolution: 5m
  - from: 90d
    to: 1y
    resolution: 1h
```

3. Compression

```
Prometheus TSDB Compression:
├── Chunk format: XOR encoding
├── Timestamp delta encoding
├── Value compression
└── Result: 1.5 bytes/sample average

Example:
Uncompressed: 16 bytes/sample
├── Timestamp: 8 bytes
└── Value: 8 bytes

Compressed: 1.5 bytes/sample
├── Timestamp delta: 0.5 bytes
└── XOR value: 1.0 bytes

Savings: 90%

Storage Calculation:
├── 1M metrics × 15s scrape = 5.76B samples/day
├── Uncompressed: 5.76B × 16 bytes = 92 GB/day
├── Compressed: 5.76B × 1.5 bytes = 8.6 GB/day
└── Savings: 91% storage reduction
```

Data Flow & Ingestion Pipeline

Log Ingestion Flow

```
┌─────────────────────────────────────────────────────────────────┐
│ STEP 1: Log Generation                                          │
└─────────────────────────────────────────────────────────────────┘
Application Server:
  2025-01-08 10:30:45 ERROR [payment-service] Payment failed
  user_id=12345 order_id=67890 error="Card declined"

┌─────────────────────────────────────────────────────────────────┐
│ STEP 2: Agent Collection (Filebeat)                             │
└─────────────────────────────────────────────────────────────────┘
Filebeat:
├── Tail /var/log/app/payment.log
├── Add metadata (host, app, env)
├── Buffer in memory (10MB)
└── Send batch to Logstash (100 events)

┌─────────────────────────────────────────────────────────────────┐
│ STEP 3: Processing (Logstash)                                   │
└─────────────────────────────────────────────────────────────────┘
Logstash Pipeline:
├── Parse timestamp
```

```
├── Extract level (ERROR)
├── Extract service name
├── Parse key-value pairs
├── Add GeoIP for user location
└── Structured output:
    {
      "@timestamp": "2025-01-08T10:30:45Z",
      "level": "ERROR",
      "service": "payment-service",
      "user_id": "12345",
      "order_id": "67890",
      "error": "Card declined",
      "host": "app-server-23",
      "env": "production"
    }

┌─────────────────────────────────────────────────────────────────┐
│ STEP 4: Buffer (Kafka)                                          │
└─────────────────────────────────────────────────────────────────┘
Kafka:
├── Topic: logs-payment
├── Partition: hash(user_id) % 10
├── Replication: 3 copies
└── Retention: 7 days

┌─────────────────────────────────────────────────────────────────┐
│ STEP 5: Storage (Elasticsearch)                                 │
└─────────────────────────────────────────────────────────────────┘
Elasticsearch:
├── Index: logs-payment-2025.01.08
├── Shard: Route to shard based on doc ID
├── Indexing: Create inverted indexes
├── Refresh: Make searchable (5s delay)
└── Replica: Copy to replica shard

┌─────────────────────────────────────────────────────────────────┐
│ STEP 6: Query (Kibana)                                          │
└─────────────────────────────────────────────────────────────────┘
User searches: "level:ERROR AND service:payment-service"
├── Query coordinator node
├── Scatter to all relevant shards
├── Gather results
├── Sort and rank
└── Return to Kibana (< 1 second)
```

Latency Breakdown:

```
End-to-End Latency (log written to searchable):
├── Filebeat buffer: 1-5 seconds
├── Logstash processing: 0.5-2 seconds
├── Kafka buffering: 0.1-1 second
```

```
├── Elasticsearch indexing: 1-5 seconds (refresh interval)
└── Total: 2.6-13 seconds (acceptable for most use cases)

Real-time scenarios (critical errors):
├── Filebeat → Direct to Elasticsearch
├── Skip Kafka buffering
└── Total: ~2-5 seconds
```

Metrics Scraping Flow

```
┌─────────────────────────────────────────────────────────────────┐
│ STEP 1: Metrics Exposition                                      │
└─────────────────────────────────────────────────────────────────┘
Application exposes /metrics endpoint:
  # HELP http_requests_total Total HTTP requests
  # TYPE http_requests_total counter
  http_requests_total{method="GET",status="200"} 12345
  http_requests_total{method="POST",status="500"} 42

┌─────────────────────────────────────────────────────────────────┐
│ STEP 2: Service Discovery (Prometheus)                          │
└─────────────────────────────────────────────────────────────────┘
Prometheus discovers targets:
├── Kubernetes API: List pods with label app=payment-service
├── Result: 5 pods found
└── Target list: [10.0.1.1:9090, 10.0.1.2:9090, ...]

┌─────────────────────────────────────────────────────────────────┐
│ STEP 3: Scrape (Every 15 seconds)                               │
└─────────────────────────────────────────────────────────────────┘
Prometheus:
├── HTTP GET 10.0.1.1:9090/metrics
├── Parse Prometheus format
├── Add labels (instance, job)
├── Timestamp samples
└── Store in TSDB

┌─────────────────────────────────────────────────────────────────┐
│ STEP 4: Storage (Prometheus TSDB)                               │
└─────────────────────────────────────────────────────────────────┘
TSDB:
├── Write to WAL (durability)
├── In-memory chunks (2 hours)
├── Compress and write to disk
└── Build index for fast queries

┌─────────────────────────────────────────────────────────────────┐
│ STEP 5: Long-term Storage (Thanos/Cortex)                       │
└─────────────────────────────────────────────────────────────────┘
Thanos Sidecar:
```

```
├── Upload 2-hour blocks to S3
├── Downsample older blocks
├── Compact blocks for efficiency
└── Retention: Indefinite (with downsampling)

┌─────────────────────────────────────────────────────────────┐
│ STEP 6: Query (Grafana)                                     │
└─────────────────────────────────────────────────────────────┘
User queries: rate(http_requests_total[5m])
├── Prometheus evaluates PromQL
├── Queries local TSDB
├── Queries Thanos for historical data
├── Aggregates results
└── Returns to Grafana (< 1 second)
```

Latency Breakdown:

```
End-to-End Latency (metric emitted to queryable):
├── Scrape interval: 0-15 seconds (average 7.5s)
├── TSDB write: < 1 second
├── In-memory indexing: < 1 second
└── Total: ~8.5 seconds average (acceptable)

Real-time requirements:
├── Reduce scrape interval to 5s
├── Or use push model for critical metrics
└── Total: ~6 seconds
```

Backpressure Handling

Problem: What if downstream can't keep up?

```
Scenario: Elasticsearch cluster overloaded

Without Backpressure:
├── Filebeat keeps sending
├── Logstash buffers in memory
├── Memory fills up
├── Logstash OOM crash
└── Data loss

With Backpressure (Kafka):
├── Kafka buffers logs (disk-backed)
├── Logstash processes at sustainable rate
├── Filebeat slows down (back-pressure)
├── Application logs buffered locally if needed
└── No data loss, graceful degradation
```

Implementation:

```
Filebeat Configuration:
output.kafka:
  compression: lz4
  max_message_bytes: 1000000
  required_acks: 1
  bulk_max_size: 2048

Kafka Configuration:
retention.ms: 604800000  # 7 days
segment.bytes: 1073741824  # 1GB
compression.type: lz4
log.retention.check.interval.ms: 300000

Logstash Configuration:
pipeline.batch.size: 125
pipeline.batch.delay: 50
pipeline.workers: 4
```

Distributed System Design

Challenge 1: Clock Skew

Problem: Servers have different clock times

```
Scenario:
Server A: 10:00:00 (correct)
Server B: 10:00:05 (5 seconds fast)
Server C: 9:59:55 (5 seconds slow)

Impact:
- Logs appear out of order
- Metrics timestamps inconsistent
- Time-based aggregations wrong
```

Solutions:

Option 1: NTP Synchronization (Recommended)

```
All servers sync with NTP
├── Accuracy: ±50ms typical
├── Good enough for logging/metrics
├── Industry standard
└── Implementation: ntpd or chronyd
```

```
Configuration:
server 0.pool.ntp.org iburst
server 1.pool.ntp.org iburst
driftfile /var/lib/ntp/drift
```

Option 2: Logical Clocks

```
Use Lamport timestamps or vector clocks
├── Total ordering without physical time
├── Complex to implement
└── Use when: Causality is critical
```

Option 3: Accept Slight Skew

```
Most logging systems handle ±1 minute skew
├── Elasticsearch accepts old/future docs
├── Prometheus uses sample time, not ingest time
└── Rarely causes issues in practice
```

Challenge 2: Data Consistency

Problem: Multiple Elasticsearch nodes, eventual consistency

```
Scenario:
Write to node A → Not immediately visible on node B

Timeline:
T0: Write log to shard 1 on node A
T1: Refresh interval (5s) — Now searchable on node A
T2: Replica copies to node B (async)
T3: Searchable on node B

Gap: User might not see recent logs immediately
```

Trade-offs:

```
Stronger Consistency:
├── Reduce refresh interval (1s)
├── Wait for replicas (wait_for_active_shards)
├── Cost: Higher latency, more resources

Eventual Consistency (Default):
├── Refresh every 5s
├── Async replication
```

```
├── Benefit: Better performance, lower cost
├── Trade-off: 5-10s delay for new data
```

Decision: Accept eventual consistency for logs

- Logs are inherently time-delayed anyway
- 5-10 second delay acceptable
- Performance benefits significant

Challenge 3: Split Brain

Problem: Network partition creates two master nodes

```
Elasticsearch Cluster:
Normal:
[Master] ←→ [Data1] ←→ [Data2]

Network Partition:
[Master] ←✗→ [Data1] ←→ [Data2]
                      ↑ Elects new master!

Result: Two masters, diverging data
```

Solution: Quorum-based election

```
Minimum Master Nodes = (total_masters / 2) + 1

Example with 3 master-eligible nodes:
├── Quorum: (3 / 2) + 1 = 2
├── Side with 2+ nodes becomes master
├── Side with <2 nodes can't elect master
└── Prevents split brain

Configuration:
discovery.zen.minimum_master_nodes: 2
```

Challenge 4: Hot Shards

Problem: One shard gets all the traffic

```
Scenario: logs-payment-{date}
├── Payment service is busiest
├── Shard for payment service overloaded
├── Other shards idle
└── Unbalanced cluster
```

```
Symptoms:
├── High CPU on one node
├── Slow queries
├── Cluster appears "slow" despite spare capacity
```

Solutions:

Option 1: Better Shard Distribution

```
Instead of routing by service:
Use routing by hash(doc_id)

Result:
├── Even distribution
├── All nodes equally loaded
└── Better resource utilization
```

Option 2: More Shards

```
Increase shards for busy indexes:
logs-payment-*: 10 shards
logs-notification-*: 3 shards

Trade-off:
├── Better distribution
├── More management overhead
```

Option 3: Separate Indexes

```
Critical services → Dedicated indexes
logs-payment-* → Dedicated hot tier nodes

Benefit:
├── Isolation from noisy neighbors
├── Dedicated resources
└── SLA protection
```

Scalability & Performance

Horizontal Scaling Strategy

Elasticsearch Cluster Scaling

```
Phase 1: Small (0-100 GB/day)
├── 3 master nodes
├── 5 data nodes
├── Cost: $2,000/month
└── Handles: 100 GB logs/day

Phase 2: Medium (100-500 GB/day)
├── 3 master nodes
├── 20 data nodes (hot tier)
├── 10 data nodes (warm tier)
├── Cost: $8,000/month
└── Handles: 500 GB logs/day

Phase 3: Large (500-1TB/day)
├── 3 master nodes
├── 50 data nodes (hot tier)
├── 20 data nodes (warm tier)
├── S3 for cold tier
├── Cost: $25,000/month
└── Handles: 1 TB logs/day

Phase 4: Very Large (1TB+/day)
├── Multiple clusters (regional)
├── Federated search across clusters
├── Object storage for cold data
├── Cost: $50,000+/month
└── Handles: Multiple TB/day
```

Prometheus Scaling

```
Single Prometheus (Up to 1M samples/sec):
├── 1 Prometheus server
├── 32 GB RAM
├── Local TSDB (15 days)
└── Cost: $500/month

Prometheus Federation (1M-10M samples/sec):
├── Regional Prometheus servers (3)
├── Central federation server
├── Thanos for long-term storage
└── Cost: $3,000/month

Prometheus + Cortex (10M+ samples/sec):
├── Cortex distributed system
├── Horizontal write scaling
├── S3/GCS for long-term storage
├── Query federation
└── Cost: $10,000+/month
```

Performance Optimization

1. Query Optimization

Elasticsearch:

```
Slow Query:
{
  "query": {
    "wildcard": {
      "message": "*error*"
    }
  }
}
Problem: Full table scan
Time: 10+ seconds

Fast Query:
{
  "query": {
    "match": {
      "message": "error"
    }
  },
  "post_filter": {
    "range": {
      "@timestamp": {
        "gte": "now-1h"
      }
    }
  }
}
Improvement: Uses inverted index + time filter
Time: <100ms
```

Prometheus PromQL:

```
Slow Query:
sum(rate(http_requests_total[5m])) by (instance)
Problem: High cardinality on instance

Fast Query:
sum(rate(http_requests_total[5m])) by (job)
Improvement: Lower cardinality on job
Time reduction: 5x faster
```

2. Caching Strategy

```
Multi-Layer Cache:

L1: Browser Cache
├── Dashboard JSON
├── TTL: 30 seconds
└── Reduces API calls

L2: CDN Cache
├── Static assets (JS, CSS)
├── Dashboard definitions
└── Reduces server load

L3: Application Cache (Redis)
├── Query results
├── TTL: 5 minutes
├── Hit rate: 60-80%
└── Reduces database load

L4: Database Cache
├── Elasticsearch query cache
├── Prometheus query cache
└── Reduces disk I/O
```

3. Batch Processing

```
Single inserts (Slow):
For each log:
  INSERT INTO elasticsearch

Time: 100 logs/second

Bulk inserts (Fast):
Batch 1000 logs:
  BULK INSERT INTO elasticsearch

Time: 100,000 logs/second
Improvement: 1000x faster
```

4. Index Optimization

```
Elasticsearch Index Tuning:

Refresh Interval:
├── Default: 1s (near real-time)
├── Optimized: 30s (for bulk loading)
└── Indexing speed: 3x faster
```

```
Replica Count:
├── Indexing: 0 replicas
├── After indexing: 1 replica
└── Indexing speed: 2x faster

Force Merge:
├── After rollover to warm tier
├── Reduces segment count (50 → 1)
└── Query speed: 5x faster

Codec:
├── Hot tier: default (fast)
├── Warm tier: best_compression
└── Storage savings: 30-40%
```

Fault Tolerance & Reliability

High Availability Architecture

```
┌──────────────────────────────────────────────────────────────┐
│                        REGION 1                               │
│  ┌──────────────────────┐  ┌──────────────────────┐          │
│  │ Availability Zone A  │  │ Availability Zone B  │          │
│  │                      │  │                      │          │
│  │  ┌────────────────┐  │  │  ┌────────────────┐  │          │
│  │  │ Elasticsearch  │  │  │  │ Elasticsearch  │  │          │
│  │  │ Master + Data  │  │  │  │ Master + Data  │  │          │
│  │  └───────┬────────┘  │  │  └───────┬────────┘  │          │
│  │          │           │  │          │           │          │
│  │  ┌───────▼────────┐  │  │  ┌───────▼────────┐  │          │
│  │  │   Prometheus   │  │  │  │   Prometheus   │  │          │
│  │  │    (Active)    │  │  │  │   (Standby)    │  │          │
│  │  └────────────────┘  │  │  └────────────────┘  │          │
│  └──────────────────────┘  └──────────────────────┘          │
└──────────────────────────────────────────────────────────────┘

Elasticsearch:
├── 3 master nodes (split across AZs)
├── N data nodes (balanced across AZs)
├── 1 replica per shard (different AZ)
└── Can lose 1 AZ and stay operational

Prometheus:
├── Active-Standby setup
├── Thanos for failover
├── Can lose 1 AZ, switch to standby
└── Historical data in S3 (survives total loss)
```

Failure Scenarios

Scenario 1: Elasticsearch Node Failure

```
Normal Operation:
[Master] → [Data1] [Data2] [Data3]
          Shard A  Shard B  Shard C
          Replica  Replica  Replica

Node2 Fails:
[Master] → [Data1] [✗Data2] [Data3]
          Shard A           Shard C
          Replica B         Replica

Recovery:
1. Master detects failure (30s)
2. Promote Replica B to primary (10s)
3. Reallocate Replica A and C (5 min)
4. Cluster back to green (5-10 min)

Impact:
├── No data loss (replicas available)
├── Queries continue (other nodes serve)
├── Slight performance degradation during rebalance
└── RTO: 30-40 seconds
```

Scenario 2: Kafka Broker Failure

```
Normal Operation:
[Producer] → [Broker1] [Broker2] [Broker3]
             Leader    Replica   Replica

Broker1 Fails:
[Producer] → [✗Broker1] [Broker2] [Broker3]
                        ↑ New Leader

Recovery:
1. ZooKeeper detects failure (6s)
2. Elect new leader from replicas (2s)
3. Update metadata (2s)
4. Producers/consumers reconnect (automatic)

Impact:
├── No data loss (replication factor 3)
├── Brief pause in writes (8-10s)
├── Automatic recovery
└── RTO: ~10 seconds
```

Scenario 3: Prometheus Server Failure

```
Active-Standby Setup:
[Active Prometheus] → [Thanos Sidecar] → [S3]
[Standby Prometheus] → [Thanos Sidecar] → [S3]

Active Fails:
1. Standby starts scraping (manual or auto)
2. Queries redirect to standby
3. Historical data served from S3 (Thanos)

Impact:
├── Gap in metrics (duration of failover)
├── Historical data intact
├── RTO: 1-5 minutes (manual) or 30s (auto)
└── RPO: Scrape interval (15s)

Mitigation:
Use Cortex for automatic failover:
├── Multiple Prometheus instances scrape
├── All write to central Cortex
├── No single point of failure
└── RTO: 0 (no downtime)
```

Data Durability

Logs (Elasticsearch):

```
Durability Mechanisms:
├── Translog (write-ahead log)
├── Replication (1+ replicas)
├── Snapshots to S3 (daily)
└── Cross-region replication (optional)

Failure Tolerance:
├── Can lose N-1 replicas
├── Translog protects unflushed writes
├── Snapshots for disaster recovery
└── RPO: < 5 seconds (translog)
```

Metrics (Prometheus):

```
Durability Mechanisms:
├── WAL (write-ahead log)
├── 2-hour blocks persisted to disk
├── Thanos uploads to S3
└── Downsampled for long-term retention
```

```
Failure Tolerance:
├── WAL protects recent data
├── S3 for historical data
├── Downsampling preserves trends
└── RPO: Scrape interval (15s)
```

Disaster Recovery

Backup Strategy:

```
Elasticsearch:
├── Daily snapshots to S3
├── Retention: 30 days
├── Cross-region replication
├── Restore time: Hours to TB+
└── Test restores monthly

Prometheus:
├── Continuous upload to S3 (Thanos)
├── No backups needed (immutable blocks)
├── Downsampling preserves data
└── Restore: Point queries at S3

Configuration:
├── Git repository (versioned)
├── Automated deployment
└── Restore: Minutes
```

Recovery Procedures:

```
Complete Cluster Loss:

Elasticsearch:
1. Provision new cluster
2. Restore latest snapshot from S3
3. Reconfigure applications
4. Time: 2-4 hours

Prometheus:
1. Provision new Prometheus
2. Configure Thanos to query S3
3. Historical data immediately available
4. Time: 15-30 minutes

Prevention:
├── Multi-region deployment
```

```
├── Regular DR drills
└── Automated runbooks
```

Multi-Region Architecture

Global Deployment

```
                    [Global DNS / Load Balancer]
                               │
              ┌────────────────┼────────────────┐
              │                │                │
              ▼                ▼                ▼
    ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
    │   US-WEST    │ │   US-EAST    │ │   EU-WEST    │
    │              │ │              │ │              │
    │ Filebeat (5K)│ │ Filebeat (3K)│ │ Filebeat (2K)│
    │      ↓       │ │      ↓       │ │      ↓       │
    │  Logstash    │ │  Logstash    │ │  Logstash    │
    │      ↓       │ │      ↓       │ │      ↓       │
    │ Elasticsearch│ │ Elasticsearch│ │ Elasticsearch│
    │   (Local)    │ │   (Local)    │ │   (Local)    │
    │              │ │              │ │              │
    │  Prometheus  │ │  Prometheus  │ │  Prometheus  │
    │      ↓       │ │      ↓       │ │      ↓       │
    │  Thanos Query│◄┤  Thanos Query├─►  Thanos Query│
    │              │ │              │ │              │
    └──────┬───────┘ └──────┬───────┘ └──────┬───────┘
           │                │                │
           └────────────────┼────────────────┘
                           ↓
                 [S3 Global Storage]
                 (Long-term, Federated)
```

Regional Independence

Design: Each region operates independently

```
Logs:
├── Stored locally in each region
├── Cross-region search via federation (optional)
├── Compliance: Data stays in region
└── No cross-region latency for queries

Metrics:
├── Scraped locally by Prometheus
├── Uploaded to global S3 (Thanos)
├── Federated queries across regions
```

```
└── Global dashboards via Thanos Query

Benefits:
✓ Data locality (faster, compliance)
✓ Regional failures isolated
✓ No cross-region bandwidth costs
✓ Simple to operate

Challenges:
✗ Global queries more complex
✗ Data scattered across regions
```

Federated Queries

Cross-Region Log Search:

```
Kibana (US-WEST) needs to search EU logs:

Option 1: Cross-Cluster Search (Elasticsearch)
┌─────────────────────────────────────────┐
│  Kibana (US-WEST)                       │
│       ↓                                 │
│  Elasticsearch (US-WEST)                │
│       ├─→ Local query                   │
│       └─→ Remote query → ES (EU-WEST)   │
│       ↓                                 │
│  Merge results                          │
└─────────────────────────────────────────┘

Latency: +100-200ms for cross-region

Option 2: Separate Queries (Recommended)
- Query each region separately
- User selects region in UI
- Faster, simpler, region-aware
```

Global Metrics Dashboard:

```
Thanos Architecture:

Regional Prometheus → Thanos Sidecar → S3
   (US-WEST)              ↓
                     [Upload blocks]

Regional Prometheus → Thanos Sidecar → S3
   (US-EAST)              ↓
                     [Upload blocks]
```

```
Regional Prometheus → Thanos Sidecar → S3
   (EU-WEST)              ↓
                     [Upload blocks]

            ↓ Query from any region
     [Thanos Query Frontend]
            ↓
     Aggregate global metrics

Benefits:
✓ Single pane of glass
✓ Global queries (sum across regions)
✓ Historical data centralized
✓ No Prometheus federation complexity
```

Design Trade-offs

1. Push vs Pull

Aspect	Push (Logs)	Pull (Metrics)
Complexity	Higher (need agents)	Lower (server-side)
Flexibility	Can send anytime	Fixed intervals
Firewalls	Works behind firewall	Needs open port
Load	Can overwhelm receiver	Controlled by scraper
Short-lived	Perfect for Lambda/cron	Needs push gateway
Service Discovery	Not needed	Built-in

Decision:

- Logs: Push (applications generate continuously)
- Metrics: Pull (Prometheus model, with push gateway for edge cases)

2. Structured vs Unstructured Logs

```
Unstructured:
"2025-01-08 10:30:45 ERROR Payment failed for user 12345"

Pros:
✓ Human readable
✓ No schema needed
✓ Flexible format

Cons:
✗ Hard to parse
```

```
✗ Slow to query
✗ Requires grok patterns

Structured (JSON):
{
  "timestamp": "2025-01-08T10:30:45Z",
  "level": "ERROR",
  "message": "Payment failed",
  "user_id": 12345
}

Pros:
✓ Fast to query
✓ Easy to aggregate
✓ Type-safe

Cons:
✗ More verbose
✗ Requires schema
✗ Less human-readable

**Decision**: Structured (JSON) for new applications
- Better query performance
- Easier aggregation
- Worth the verbosity
- Can always add unstructured "message" field
```

3. Centralized vs Distributed Storage

```
Centralized (Single Cluster):
All regions → Central Elasticsearch

Pros:
✓ Simple operations
✓ Easy to query
✓ Single source of truth

Cons:
✗ Cross-region latency
✗ Single point of failure
✗ Compliance issues
✗ Expensive bandwidth

Distributed (Regional Clusters):
Each region → Local Elasticsearch

Pros:
✓ Data locality
✓ Failure isolation
✓ Compliance friendly
```

```
✓ Lower latency

Cons:
✗ Complex operations
✗ Harder to query globally
✗ More clusters to manage

**Decision**: Distributed for large scale
- Centralized for small deployments (<100 GB/day)
- Distributed for large deployments
- Use federation for cross-region queries
```

4. Retention Policy

```
Option 1: Single Tier (Simple)
├── All data in Elasticsearch
├── Retention: 30 days
├── Cost: High (all on SSD)
└── Query: Fast

Option 2: Multi-Tier (Recommended)
├── Hot: 7 days (SSD)
├── Warm: 30 days (slower SSD)
├── Cold: 90 days (S3)
├── Archive: 1 year (Glacier)
├── Cost: Medium (tiered)
└── Query: Fast for recent, slower for old

Option 3: Immediate Archive
├── All data to S3 immediately
├── Elasticsearch indexes only recent
├── Query: Slow (S3 every time)
└── Cost: Low

**Decision**: Multi-tier
- Balance of cost and performance
- 90% queries hit hot/warm tier
- 10% queries slower (acceptable)
- 80% cost savings vs single tier
```

Real-World Case Studies

Case Study 1: Netflix (1000+ Microservices)

Scale:

- 10,000+ EC2 instances
- 100+ TB logs/day

- 2.5 billion metrics per day
- 500+ petabytes historical data

Architecture:

```
Logging Stack:
├── Agent: Vector, Fluentd
├── Processing: Apache Flink
├── Storage: Elasticsearch (primary), S3 (archive)
├── Query: Kibana, custom tools
└── Retention: 30 days hot, years in S3

Metrics Stack:
├── Collection: Atlas (custom), Prometheus
├── Storage: Atlas TSDB
├── Processing: Spark for analytics
├── Visualization: Grafana, custom dashboards
└── Alerting: Custom alerting infrastructure
```

Key Decisions:

1. **Atlas (Custom TSDB)**: Built for their specific needs
    - Optimized for their workload (dimensional time-series)
    - Integration with internal systems
    - Better performance than off-the-shelf
2. **Elasticsearch for logs**: Industry standard, proven at scale
    - Good search performance
    - Rich ecosystem
    - Multi-tenancy support
3. **S3 for archival**: Cost-effective long-term storage
    - Compliance requirements
    - Rarely queried historical data
    - 95% cost savings vs hot storage

Lessons Learned:

- Build custom tools only when necessary (Atlas for specific needs)
- Use proven solutions where possible (Elasticsearch)
- Invest in automation (alert management, capacity planning)
- Sampling is key at extreme scale (sample 1% of logs for analytics)

Case Study 2: Uber (Global Ride-Sharing)

Scale:

- 50,000+ microservices
- Millions of events per second
- Petabyte-scale storage
- Real-time analytics

Architecture:

```
Logging Stack (M3):
├── Collection: Kafka
├── Processing: Apache Flink
├── Storage: M3DB (custom TSDB)
├── Query: Custom query engine
└── Visualization: Grafana

Key Innovation: M3DB
├── Distributed time-series database
├── Built for extreme scale
├── Prometheus-compatible
├── Global aggregations
└── Open-sourced
```

Design Highlights:

1. **M3DB Architecture**:
    - Namespaces for different retention periods
    - Aggressive compression
    - Distributed aggregation
    - Multi-datacenter replication
2. **Real-time Processing**:
    - Flink for stream processing
    - Sub-second latency
    - Exactly-once semantics
3. **Cost Optimization**:
    - Downsampling (1s → 1m → 5m)
    - Compression (6:1 ratio)
    - Tiered storage

Lessons Learned:

- Prometheus doesn't scale to millions of metrics (needs custom solution)
- Real-time anomaly detection is critical for ride-sharing
- Global aggregation necessary for business metrics
- Investment in custom tooling pays off at scale

Case Study 3: Datadog (SaaS Observability)

Scale:

- Hundreds of billions of metrics per day
- Petabytes of logs per day
- Thousands of customers
- 99.99% SLA

Architecture:

```
Multi-Tenant Design:
├── Data isolation per customer
├── Shared infrastructure
├── Usage-based pricing
├── Rate limiting per tenant
└── SLA guarantees per tier

Storage:
├── Hot tier: Custom TSDB
├── Cold tier: S3
├── Real-time: Apache Kafka
├── Analytics: Apache Spark
└── Retention: Configurable (1 day - 15 months)

Querying:
├── Custom query language
├── Distributed query execution
├── Query optimization
└── Result caching
```

Key Innovations:

1. **Multi-tenancy at scale**: Thousands of isolated customers on shared infrastructure
2. **Intelligent sampling**: Sample to reduce costs while maintaining accuracy
3. **Unified platform**: Metrics, logs, traces in single platform
4. **ML-powered alerts**: Reduce false positives

Lessons Learned:

- Multi-tenancy requires careful resource isolation
- Query optimization critical for SaaS performance
- Sampling strategies must be transparent to customers
- Rich integrations drive adoption (500+ integrations)

Case Study 4: LinkedIn (Professional Network)

Scale:

- 20,000+ hosts
- 1.5+ trillion metrics per day
- 300+ TB logs per day
- Real-time analytics

Architecture:

```
Logging Stack (Kafka + Samza):
├── Collection: Kafka
├── Processing: Apache Samza
├── Storage: Kafka compacted topics
├── Search: Elasticsearch
└── Analytics: Pinot

Metrics Stack (InGraphs):
├── Collection: Custom agents
├── Storage: Custom TSDB (Venice)
├── Processing: Spark
├── Query: Custom query engine
└── Visualization: Custom dashboards
```

Key Innovations:

1. **Kafka as storage**: Use Kafka for both transport and storage
2. **Venice TSDB**: Distributed key-value store optimized for time-series
3. **Pinot**: Real-time OLAP for analytics
4. **Samza**: Stream processing for real-time aggregations

Lessons Learned:

- Kafka's durability makes it good for log storage
- Custom solutions needed at extreme scale
- Real-time aggregation reduces storage costs
- Query federation across systems for flexibility

Technology Stack Comparison

Logging Stacks

Component	ELK Stack	EFK Stack	Splunk	Loki
Agent	Filebeat	Fluentd	Universal Forwarder	Promtail
Processing	Logstash	Fluentd	Splunk Indexers	None
Storage	Elasticsearch	Elasticsearch	Splunk	Object Storage
Query	Kibana	Kibana	Splunk Web	Grafana

Component	ELK Stack	EFK Stack	Splunk	Loki
Cost	Medium	Medium	High	Low
Scale	High	High	Very High	Medium
Ease of Use	Medium	Medium	High	High

Recommendations:

- **ELK/EFK**: General purpose, proven at scale
- **Splunk**: Enterprise features, high cost
- **Loki**: Cost-conscious, simple log aggregation

Metrics Stacks

Component	Prometheus	Graphite	InfluxDB	Datadog
Collection	Pull	Push	Push	Agent
Storage	Local TSDB	Whisper	TSM Engine	Cloud
Query Language	PromQL	Functions	InfluxQL/Flux	Custom
Scalability	Medium	Low	Medium	Very High
Cost	Free (OSS)	Free (OSS)	Paid	$$$
Long-term	Thanos/Cortex	Carbon Relay	InfluxDB Cloud	Built-in

Recommendations:

- **Prometheus**: Cloud-native, Kubernetes, OSS
- **InfluxDB**: IoT, high cardinality
- **Datadog**: SaaS, full observability platform

Interview Talking Points

Key Design Decisions

1. Why Elasticsearch for logs?

- Full-text search capability
- Flexible schema (JSON documents)
- Horizontal scalability
- Rich aggregation engine
- Industry standard with large ecosystem

2. Why Prometheus for metrics?

- Pull-based model (better for service discovery)
- Efficient time-series storage (1.5 bytes/sample)

- Powerful query language (PromQL)
- Active community, CNCF graduated
- Kubernetes-native

3. Why Kafka as message queue?

- High throughput (millions events/sec)
- Durability (replicated, disk-backed)
- Replay capability for debugging
- Multiple consumers (fanout pattern)
- Decouples producers from consumers

4. Why separate hot/warm/cold tiers?

- 80% cost reduction
- 90% queries hit recent data (hot tier)
- Compliance needs long-term retention
- Query performance optimized for common case

5. Why not store everything in real-time databases?

- Cost prohibitive (SSD expensive)
- Most queries are recent data
- Historical data queried rarely
- Acceptable to have slower queries for old data

6. Why multi-region deployment?

- Data locality (lower latency)
- Compliance (GDPR, data residency)
- Fault isolation (regional failures)
- Reduced cross-region bandwidth costs

Bottlenecks & Solutions

Bottleneck	Impact	Solution
Elasticsearch indexing	Slow ingestion	Bulk API, tune refresh interval
High cardinality metrics	Memory explosion	Limit label combinations, sampling
Cross-region queries	High latency	Regional clusters, federated search
Hot shards	Uneven load	Better routing, more shards
Query performance	Slow dashboards	Caching, time-range filters, indexing
Storage costs	Budget overrun	Tiered storage, downsampling, compression
Kafka lag	Processing delay	Add consumers, increase partitions
Clock skew	Out-of-order data	NTP sync, logical ordering

Scale Calculations

```
Logs:
- 10,000 servers × 1,000 lines/min = 10M lines/min
- 10M × 500 bytes = 5 GB/min = 7.2 TB/day
- 90 days retention = 648 TB

Metrics:
- 10,000 servers × 100 metrics = 1M metrics
- 1M × (86400/15) samples = 5.76B samples/day
- 5.76B × 1.5 bytes = 8.6 GB/day
- 90 days retention = 774 GB

Infrastructure:
- Elasticsearch: 50-100 nodes (10-20 TB each)
- Prometheus: 10-20 servers (500 GB each)
- Kafka: 10 brokers (5 TB each)
- Logstash: 20-30 workers
```

Common Interview Questions

Q: How do you handle 1M log events per second?
A:

- Kafka buffers incoming logs (handles spikes)
- Multiple Logstash workers process in parallel
- Bulk insert to Elasticsearch (1000 docs at once)
- Tune Elasticsearch refresh interval (5s → 30s)
- Use ingest nodes for heavy processing
- Scale horizontally by adding nodes

Q: How do you prevent high-cardinality metrics from causing issues?
A:

- Design guidelines (labels <100 unique values)
- Automated validation (reject metrics with too many labels)
- Sampling for high-cardinality (1 in 100)
- Use aggregations in recording rules
- Alert on cardinality explosion

Q: What happens when Elasticsearch cluster is full?
A:

- Index Lifecycle Management (ILM) automatically archives old data
- Disk watermarks prevent complete fill (85%, 90%, 95%)
- At 95%: Block new writes to protect cluster
- Solution: Add nodes or delete old data
- Prevention: Capacity planning, alerts at 70%

Q: How do you ensure no log data is lost?
A:

- Filebeat persists position (knows what was sent)
- Kafka replication (3 copies)
- Elasticsearch translog (write-ahead log)
- Elasticsearch replicas (1-2 copies)
- S3 snapshots (disaster recovery)
- Result: Multiple layers of durability

Q: How do you handle time-series data with high cardinality?
A:

- Don't use labels for high-cardinality dimensions
- Use relabeling to drop or aggregate labels
- Implement recording rules (pre-aggregation)
- Use Thanos/Cortex for horizontal scaling
- Consider alternative databases (ClickHouse, TimescaleDB)

Q: How would you reduce costs by 50%?
A:

- Aggressive tiering (hot → warm → cold)
- Sampling (logs: 10%, metrics: downsampling)
- Compression (best_compression codec)
- Right-size clusters (don't over-provision)
- Delete unnecessary data
- Use object storage for archives

Q: How do you ensure 99.99% uptime?
A:

- Multi-AZ deployment
- Replication (all data has 2+ copies)
- Automatic failover (Elasticsearch, Kafka)
- Health checks and circuit breakers
- Graceful degradation (serve cached data)
- Regular disaster recovery drills

System Design Patterns Used

1. **Producer-Consumer Pattern**

- Applications produce logs/metrics
- Kafka queues decouple producers from consumers
- Multiple consumers process at their own pace
- Enables backpressure handling

2. **Fan-Out Pattern**

- Single log stream → Multiple consumers
- Elasticsearch for search
- S3 for archival
- Spark for analytics
- Independent processing

3. **CQRS (Command Query Responsibility Segregation)**

- Separate write path (ingestion) from read path (query)
- Optimize each independently
- Write: Bulk inserts, batching
- Read: Indexes, caching, replicas

4. **Circuit Breaker**

- Protect downstream services from overload
- Fail fast when service is down
- Automatic recovery when service returns
- Implemented in Filebeat, Logstash

5. **Bulkhead Pattern**

- Isolate resources for different services
- Separate indexes for critical services
- Prevent noisy neighbors
- Dedicated thread pools

6. **Time-Series Compaction**

- Prometheus 2-hour blocks
- Compact older blocks
- Reduce storage and improve query speed
- Immutable blocks after compaction

7. **Tiered Storage**

- Hot tier (SSD) for recent data
- Warm tier (HDD) for older data
- Cold tier (S3) for archives
- Automatic lifecycle management

8. **Sampling**

- Sample high-volume logs (1 in 100)
- Downsample metrics for long-term storage
- Maintain accuracy for rare events
- Reduce costs significantly

Best Practices & Recommendations

Log Best Practices

1. Structured Logging

```
// Good: Structured JSON
{
  "timestamp": "2025-01-08T10:30:45Z",
  "level": "ERROR",
  "service": "payment-service",
  "trace_id": "abc123",
  "user_id": 12345,
  "error": "Card declined",
  "error_code": "INSUFFICIENT_FUNDS"
}

// Bad: Unstructured string
"2025-01-08 10:30:45 ERROR Payment failed for user 12345 card declined"
```

2. Log Levels

```
DEBUG:   Detailed diagnostic information
INFO:    General informational messages
WARNING: Warning messages (not errors)
ERROR:   Error messages (caught exceptions)
FATAL:   Critical errors (system shutdown)

Best practice:
- Production: INFO and above
- Staging: DEBUG
- Never log sensitive data (passwords, tokens, PII)
```

3. Correlation IDs

```
Add trace_id to all logs in request path:
├── API Gateway: Generate trace_id
├── All downstream services: Propagate trace_id
├── All logs: Include trace_id
└── Benefit: Track request across services

Example:
trace_id: "req-abc123"
├── API Gateway: "Received request"
├── Auth Service: "User authenticated"
```

```
├── Payment Service: "Payment processed"
└── Can correlate entire request flow
```

Metrics Best Practices

1. Naming Convention

```
Format: {namespace}_{subsystem}_{name}_{unit}

Good examples:
http_requests_total
http_request_duration_seconds
node_cpu_seconds_total
api_errors_total

Bad examples:
requests  (too generic)
my_metric (no context)
time_ms (non-standard unit)
```

2. Label Best Practices

```
Good labels:
http_requests_total{method="GET", status="200", endpoint="/api/users"}

Bad labels:
http_requests_total{user_id="12345", request_id="abc"}
└── High cardinality causes problems

Rule:
- Use labels for dimensions you'll query by
- Keep cardinality low (<1000 combinations)
- Document label values
```

3. The Four Golden Signals (Google SRE)

```
1. Latency: How long requests take
   http_request_duration_seconds

2. Traffic: How much demand
   http_requests_total

3. Errors: Rate of failed requests
   http_requests_total{status="5xx"}

4. Saturation: How "full" the service is
```

```
   - CPU usage
   - Memory usage
   - Disk I/O
   - Queue depth
```

4. RED Method (Requests, Errors, Duration)

```
Requests: Rate of requests
  rate(http_requests_total[5m])

Errors: Rate of errors
  rate(http_requests_total{status="5xx"}[5m])

Duration: Latency distribution
  histogram_quantile(0.99,
    rate(http_request_duration_seconds_bucket[5m])
  )
```

Alert Best Practices

1. Alert on Symptoms, Not Causes

```
Good:
alert: HighErrorRate
expr: error_rate > 0.01
description: "Users experiencing errors"

Bad:
alert: HighCPU
expr: cpu > 80%
description: "CPU is high"
└── CPU high might not affect users
```

2. Alert Fatigue Prevention

```
Strategies:
├── Set appropriate thresholds (not too sensitive)
├── Require duration (for: 5m)
├── Group related alerts
├── Use inhibition rules
├── Meaningful alert names and descriptions
└── Runbooks for each alert

Example:
alert: DatabaseDown
expr: up{job="postgres"} == 0
```

```
for: 1m  # Not just a blip
annotations:
  runbook: "https://wiki.company.com/db-down"
  action: "Check database logs, attempt restart"
```

3. Alert Severity Levels

```
Critical (Page immediately):
├── Service is down
├── Data loss occurring
├── Security breach
└── Customer impact

Warning (Notify, don't page):
├── Degraded performance
├── Approaching capacity
├── Non-critical error rate increase
└── Can wait for business hours

Info (Log only):
├── Successful deployments
├── Scaling events
├── Routine maintenance
```

Cost Optimization Strategies

Storage Cost Reduction

1. Tiered Storage

```
Cost comparison (per TB/month):
├── SSD (hot): $100
├── HDD (warm): $40
├── S3 Standard: $23
├── S3 IA: $12.50
└── S3 Glacier: $4

Strategy:
├── 7 days on SSD: 50 TB × $100 = $5,000
├── 30 days on HDD: 200 TB × $40 = $8,000
├── 90 days on S3: 600 TB × $23 = $13,800
├── 1 year on Glacier: 2.5 PB × $4 = $10,000
└── Total: $36,800/month vs $100,000 (all SSD)

Savings: 63%
```

2. Compression

```
Elasticsearch:
├── Default codec: ~30% compression
├── Best_compression: ~50% compression
├── Trade-off: Slightly slower indexing
└── Use for warm tier

Prometheus:
├── XOR encoding: ~90% compression
├── Automatic, no configuration needed
└── 92 GB → 8.6 GB per day
```

3. Sampling

```
Use cases for sampling:
├── Debug logs: Sample 10% (still useful)
├── Metrics: Downsample after retention period
├── Traces: Sample 1-5% of requests
└── Analytics: Statistical sampling sufficient

Example:
100 GB/day → Sample 10% → 10 GB/day
Savings: 90% ($2,300 → $230/month)
```

4. Retention Optimization

```
Analyze query patterns:
├── 90% queries: Last 7 days
├── 9% queries: Last 30 days
├── 1% queries: Older than 30 days

Optimize retention:
├── Hot (7 days): Fast, expensive
├── Warm (30 days): Medium speed/cost
├── Cold (90+ days): Slow, cheap
└── Delete or archive after 1 year

Result: 70-80% cost reduction
```

Compute Cost Reduction

1. Right-Sizing

```
```

```
Monitor actual usage:
├── CPU utilization: Target 60-70%
├── Memory utilization: Target 70-80%
├── Disk I/O: Target 60-70%
└── Right-size instances accordingly

Example:
├── Over-provisioned: c5.4xlarge (16 vCPU, $0.68/hr)
├── Right-sized: c5.2xlarge (8 vCPU, $0.34/hr)
└── Savings: 50%
```

2. Auto-Scaling

```
Scale based on metrics:
├── Scale up: CPU > 70% for 5 minutes
├── Scale down: CPU < 30% for 15 minutes
├── Min instances: Baseline load
├── Max instances: Peak load + buffer
└── Savings: 30-40% vs static provisioning
```

3. Spot Instances

```
Use cases:
├── Logstash workers (fault-tolerant)
├── Batch processing jobs
├── Non-critical workloads
└── Savings: 60-80%

Not recommended for:
✗ Elasticsearch data nodes (state)
✗ Prometheus (metrics loss)
✗ Kafka brokers (data loss risk)
```

Security & Compliance

Access Control

Authentication:

```
Options:
├── LDAP/Active Directory integration
├── SAML/OAuth for SSO
├── API keys for programmatic access
├── JWT tokens for service-to-service
```

```
└── MFA for admin access

Implementation:
Kibana → Elasticsearch:
  xpack.security.enabled: true
  xpack.security.authc.providers:
    saml.saml1:
      order: 0
      realm: saml-realm
```

Authorization:

```
Role-Based Access Control (RBAC):

Roles:
├── Admin: Full access (create, delete, configure)
├── Developer: Read/write logs and metrics
├── Viewer: Read-only access
├── Service: API access for applications
└── Auditor: Read-only, compliance reports

Implementation:
- Elasticsearch: Index-level permissions
- Prometheus: External authentication proxy
- Kibana: Space-based isolation (multi-tenancy)
- Grafana: Folder and dashboard permissions
```

Audit Logging

```
Track all actions:
├── Who: User or service account
├── What: Action performed (read, write, delete)
├── When: Timestamp
├── Where: Resource accessed
├── Result: Success or failure

Implementation:
- Elasticsearch audit logs
- Store in separate index
- Retention: 1+ years (compliance)
- Immutable (append-only)
- Monitored for anomalies

Example audit log:
{
  "user": "john.doe@company.com",
  "action": "DELETE_INDEX",
  "resource": "logs-payment-2024.12.01",
```

```
  "timestamp": "2025-01-08T10:30:45Z",
  "result": "SUCCESS",
  "ip": "192.168.1.1"
}
```

Data Privacy

PII Handling:

```
Best practices:
├── Don't log PII (passwords, SSN, credit cards)
├── Hash sensitive fields (email, user_id)
├── Mask sensitive data (show last 4 digits)
├── Separate indexes for sensitive data
└── Encryption at rest

Implementation:
Logstash filter:
filter {
  # Mask credit card numbers
  mutate {
    gsub => [
      "message", "\d{4}-\d{4}-\d{4}-(\d{4})",
      "****-****-****-\1"
    ]
  }

  # Hash email addresses
  fingerprint {
    source => "email"
    target => "email_hash"
    method => "SHA256"
  }

  # Remove original email
  mutate {
    remove_field => ["email"]
  }
}
```

Encryption:

```
In Transit:
├── TLS 1.3 for all communications
├── Certificate validation
├── mTLS for service-to-service
└── VPN for cross-region
```

```
At Rest:
├── Elasticsearch: Encryption at rest (EBS encryption)
├── S3: Server-side encryption (SSE-S3, SSE-KMS)
├── Snapshots: Encrypted
└── Kafka: Encryption at rest (optional)
```

Compliance

GDPR/Data Residency:

```
Requirements:
├── Data must stay in EU for EU users
├── Right to access (export user's logs)
├── Right to be forgotten (delete user's logs)
└── Data processing agreements

Implementation:
├── Regional clusters (no cross-border transfers)
├── User data export API
├── Automated deletion workflows
├── Index lifecycle management
└── Audit logging of all access
```

Retention Policies:

```
By data type:
├── Application logs: 90 days
├── Security logs: 1 year (compliance)
├── Audit logs: 7 years (legal requirement)
├── Debug logs: 7 days (space-saving)
└── Metrics: 15 days raw, 1 year downsampled

Automated enforcement:
- ILM policies in Elasticsearch
- Thanos compaction and downsampling
- S3 lifecycle policies
- Regular compliance audits
```

Technology Stack Summary

Layer	Logging	Metrics	Purpose
Collection	Filebeat, Fluentd	Prometheus, Telegraf	Gather data
Transport	Kafka	Kafka (optional)	Buffer, decouple

Layer	Logging	Metrics	Purpose
Processing	Logstash, Flink	Recording rules	Transform, enrich
Storage (Hot)	Elasticsearch	Prometheus TSDB	Fast queries
Storage (Cold)	S3, Glacier	Thanos, Cortex	Long-term, cheap
Query	Elasticsearch DSL	PromQL	Retrieve data
Visualization	Kibana	Grafana	Dashboards
Alerting	ElastAlert, Watcher	Alertmanager	Notifications
Analysis	Spark, Flink	Spark	Batch analytics

Summary & Key Takeaways

Critical Design Principles

1. **Separation of Concerns**
    - Separate ingestion, storage, and query layers
    - Each can scale independently
    - Failures isolated
2. **Eventual Consistency is Acceptable**
    - Logs don't need strong consistency
    - 5-10 second delay for searchability is fine
    - Allows for better performance and cost
3. **Push for Logs, Pull for Metrics**
    - Logs: Continuous generation, need push
    - Metrics: Periodic collection, pull works better
    - Hybrid approach for edge cases
4. **Tiered Storage is Essential**
    - 80% cost savings
    - Query patterns favor recent data
    - Compliance needs long-term retention
5. **Kafka is the Universal Buffer**
    - Decouples producers from consumers
    - Handles backpressure
    - Enables replay for debugging
    - Multiple consumers without overhead
6. **Sampling Maintains Accuracy**

- Statistical sampling (1-10%)
- Maintains trends and patterns
- Massive cost savings
- Use for high-volume, low-value data

Scaling Strategy

```
Phase 1: Single Server (MVP)
├── Single Elasticsearch node
├── Single Prometheus
├── Filebeat + Logstash on same hosts
├── Cost: $500/month
└── Scale: Up to 10 hosts

Phase 2: Small Cluster
├── 3-node Elasticsearch
├── Kafka for buffering
├── Prometheus with local storage
├── Cost: $2,000/month
└── Scale: Up to 100 hosts

Phase 3: Regional Deployment
├── 20+ node Elasticsearch
├── Kafka cluster (3+ brokers)
├── Prometheus federation
├── S3 for archives
├── Cost: $10,000/month
└── Scale: Up to 1,000 hosts

Phase 4: Multi-Region
├── Regional Elasticsearch clusters
├── Thanos for global metrics
├── Federated search
├── Cost: $50,000+/month
└── Scale: 10,000+ hosts
```

Common Pitfalls to Avoid

❌ **Not planning for growth**

- Start with scalable architecture
- Don't paint yourself into corner

❌ **Logging everything**

- Log what's needed, not everything
- Sampling for high-volume

❌ **High-cardinality labels**

- Explosion of time-series
- Memory and disk issues

❌ **No retention policy**

- Storage costs spiral
- Set and enforce policies

❌ **Single point of failure**

- Always replicate
- Multi-AZ deployment

❌ **Ignoring query performance**

- Optimize indexes
- Use time range filters
- Cache frequently accessed data

❌ **No disaster recovery plan**

- Regular backups
- Test restores
- Document procedures

❌ **Alert fatigue**

- Too many alerts = ignored alerts
- Focus on actionable alerts
- Meaningful thresholds

Interview Checklist

Before the Interview

- [ ] Understand the three pillars of observability (logs, metrics, traces)
- [ ] Know ELK stack architecture and components
- [ ] Understand Prometheus architecture and TSDB
- [ ] Review time-series compression techniques
- [ ] Study distributed systems challenges (clock skew, consistency)
- [ ] Understand storage tiering strategies
- [ ] Know common query patterns and optimizations
- [ ] Review real-world case studies (Netflix, Uber, etc.)

During the Interview

1. Clarify Requirements (5 min)

- [ ] Scale (how many servers, logs/day, metrics)

- [ ] Query patterns (real-time vs historical)
- [ ] Retention requirements (hot vs cold)
- [ ] Budget constraints

2. Capacity Estimation (5 min)

- [ ] Log volume (lines/sec, bytes/line)
- [ ] Metrics volume (samples/sec, cardinality)
- [ ] Storage requirements (hot + cold)
- [ ] Infrastructure sizing

3. High-Level Design (15 min)

- [ ] Draw architecture diagram
- [ ] Identify major components
- [ ] Explain data flow
- [ ] Discuss technology choices

4. Deep Dives (20 min)

- [ ] Ingestion pipeline (buffering, backpressure)
- [ ] Storage architecture (sharding, replication)
- [ ] Query optimization
- [ ] Alerting system
- [ ] Cost optimization

5. Scalability (5 min)

- [ ] Horizontal scaling approach
- [ ] Handling growth (10x, 100x)
- [ ] Multi-region strategy

6. Fault Tolerance (5 min)

- [ ] Failure scenarios and handling
- [ ] Disaster recovery plan
- [ ] Data durability guarantees

7. Trade-offs (5 min)

- [ ] Discuss alternatives considered
- [ ] Identify potential bottlenecks
- [ ] Cost vs performance trade-offs

Further Reading

Essential Resources

Observability Fundamentals:

- "Distributed Systems Observability" - Cindy Sridharan
- "Site Reliability Engineering" - Google SRE Book
- "Observability Engineering" - Charity Majors, Liz Fong-Jones, George Miranda

Elastic Stack:

- Elasticsearch: The Definitive Guide
- ELK Stack Documentation - elastic.co/guide
- Elastic Blog - elastic.co/blog

Prometheus & Monitoring:

- "Prometheus: Up & Running" - Brian Brazil
- Prometheus Documentation - prometheus.io/docs
- "Monitoring with Prometheus" - James Turnbull

Distributed Systems:

- "Designing Data-Intensive Applications" - Martin Kleppmann
- "Database Internals" - Alex Petrov
- "The Art of Scalability" - Martin Abbott

Company Engineering Blogs

Netflix:

- Atlas: Custom monitoring system
- Mantis: Real-time event stream processing
- Edgar: Log ingestion and processing

Uber:

- M3: Distributed metrics platform
- Jaeger: Distributed tracing (now CNCF)
- Real-time analytics with Apache Flink

LinkedIn:

- Venice: Distributed key-value store
- Pinot: Real-time OLAP datastore
- Kafka as log storage

Datadog:

- Multi-tenant observability at scale
- Query optimization techniques
- Sampling strategies

Tools & Technologies

Log Management:

- ELK Stack (Elasticsearch, Logstash, Kibana)
- EFK Stack (Elasticsearch, Fluentd, Kibana)
- Splunk Enterprise
- Grafana Loki
- Graylog

Metrics & Monitoring:

- Prometheus + Grafana
- InfluxDB + Chronograf
- Datadog (SaaS)
- New Relic (SaaS)
- SignalFx (SaaS)

Time-Series Databases:

- Prometheus TSDB
- InfluxDB
- TimescaleDB
- M3DB
- VictoriaMetrics

Message Queues:

- Apache Kafka
- Amazon Kinesis
- RabbitMQ
- Apache Pulsar

Appendix

Key Metrics Formulas

Storage Calculation:

```
Log Storage = Events/sec × Avg Size × Seconds/day × Retention Days
            = 167,000 × 500 bytes × 86,400 × 90
            = 648 TB

Metric Storage = Metrics × Samples/day × Sample Size × Retention Days
               = 1M × 5.76B × 1.5 bytes × 90
               = 774 GB (with compression)
```

Throughput Calculation:

```
Logs:
├── 10,000 servers
├── 1,000 lines/min per server
├── 10M lines/min = 167K lines/sec
└── At 500 bytes/line = 83 MB/sec

Metrics:
├── 1M metrics
├── 15 second scrape interval
├── 1M / 15 = 67K samples/sec
└── At 16 bytes/sample = 1 MB/sec
```

Cluster Sizing:

```
Elasticsearch nodes = Daily Volume GB / Node Capacity GB
                    = 7,200 GB / 150 GB per node
                    = 48 data nodes

Prometheus storage = Metrics × Samples/day × Sample Size
                   = 1M × 5.76M × 1.5 bytes
                   = 8.6 GB/day
                   = 129 GB for 15 days
```

Common Latency Numbers

```
L1 cache reference:             0.5 ns
L2 cache reference:             7 ns
Main memory reference:          100 ns
SSD random read:                150 μs
Disk seek:                      10 ms
Read 1 MB sequentially from SSD: 1 ms
Read 1 MB from disk:            20 ms
Round trip in datacenter:       0.5 ms
Round trip CA → Netherlands:    150 ms

Observability System Latencies:
├── Filebeat to Logstash:       1-5 ms
├── Logstash processing:        0.5-2 ms
├── Kafka write:                1-10 ms
├── Elasticsearch index:        1-5 seconds (refresh)
├── Prometheus scrape:          15 seconds (interval)
└── Query latency:              10-1000 ms
```

Elasticsearch Performance Tuning

```
JVM Heap Size:
├── Recommendation: 50% of RAM, max 31 GB
├── Example: 64 GB RAM → 31 GB heap
├── Reason: Compressed OOPs optimization
└── Rest of RAM for OS file cache

Shard Best Practices:
├── Shard size: 30-50 GB
├── Shards per node: <1000
├── Replicas: 1 for production
└── Over-sharding hurts performance

Indexing Performance:
├── Bulk API (1000 documents)
├── Refresh interval: 30s (vs 1s default)
├── Replica: 0 during bulk, then 1
└── Result: 10x faster indexing
```

Prometheus Best Practices

```
Cardinality Management:
├── Monitor cardinality: prometheus_tsdb_symbol_table_size_bytes
├── Alert on explosion: >10M time-series
├── Use recording rules for aggregation
└── Drop high-cardinality labels

Memory Sizing:
├── Formula: (Metrics × Cardinality × Sample Size) / Compression Ratio
├── Example: (1000 × 100 × 16 bytes) / 10 = 160 KB
├── Add overhead: × 3
└── Total: ~500 KB per metric

Scrape Interval:
├── Default: 15s (good for most cases)
├── High-frequency: 5s (critical systems)
├── Low-frequency: 60s (slow-changing metrics)
└── Trade-off: Granularity vs storage
```

Quick Reference

Elasticsearch Common Queries

```
// Search error logs in last hour
{
  "query": {
    "bool": {
```

```
      "must": [
        {"match": {"level": "ERROR"}},
        {"range": {"@timestamp": {"gte": "now-1h"}}}
      ]
    }
  }
}

// Count errors by service
{
  "size": 0,
  "aggs": {
    "by_service": {
      "terms": {
        "field": "service.keyword",
        "size": 10
      }
    }
  }
}

// Errors over time (time-series)
{
  "size": 0,
  "aggs": {
    "errors_timeline": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "5m"
      }
    }
  }
}
```

Prometheus Common Queries

```
# Request rate
rate(http_requests_total[5m])

# Error percentage
100 * sum(rate(http_requests_total{status="500"}[5m]))
    / sum(rate(http_requests_total[5m]))

# 99th percentile latency
histogram_quantile(0.99,
  sum(rate(http_request_duration_seconds_bucket[5m])) by (le)
)

# Memory usage by container
sum(container_memory_usage_bytes) by (container_name)
```

```
# Predict disk full (linear regression)
predict_linear(node_filesystem_free_bytes[1h], 4*3600) < 0
```

Alert Examples

Elasticsearch Watcher:

```
{
  "trigger": {
    "schedule": {
      "interval": "5m"
    }
  },
  "input": {
    "search": {
      "request": {
        "indices": ["logs-*"],
        "body": {
          "query": {
            "bool": {
              "must": [
                {"match": {"level": "ERROR"}},
                {"range": {"@timestamp": {"gte": "now-5m"}}}
              ]
            }
          }
        }
      }
    }
  },
  "condition": {
    "compare": {
      "ctx.payload.hits.total": {
        "gt": 100
      }
    }
  },
  "actions": {
    "notify_slack": {
      "webhook": {
        "url": "https://hooks.slack.com/...",
        "body": "High error rate detected"
      }
    }
  }
}
```

Prometheus Alertmanager:

```yaml
groups:
  - name: system_alerts
    interval: 30s
    rules:
      - alert: HighMemoryUsage
        expr: |
          (node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes)
          / node_memory_MemTotal_bytes > 0.85
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Memory usage above 85% on {{ $labels.instance }}"

      - alert: DiskWillFillIn4Hours
        expr: |
          predict_linear(node_filesystem_free_bytes[1h], 4*3600) < 0
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "Disk on {{ $labels.instance }} will fill in 4 hours"
```

Document Version: 1.0 (HLD Focus)
Last Updated: January 8, 2025
Author: System Design Interview Prep
Focus: High-Level Design, Architecture, Scalability
Status: Complete & Interview-Ready ✅

Key Takeaways:

1. **ELK Stack is industry standard** for log management at scale
2. **Prometheus** is the de facto metrics system for cloud-native applications
3. **Kafka provides crucial buffering** and decouples producers from consumers
4. **Tiered storage** saves 70-80% on costs while maintaining performance
5. **Push model for logs**, pull model for metrics (with exceptions)
6. **Structured logging (JSON)** enables faster queries and aggregations
7. **Low-cardinality labels** are critical for metrics performance
8. **Multi-region deployment** provides data locality and fault isolation
9. **Downsampling and compression** enable long-term metric retention
10. **Eventual consistency is acceptable** for observability data

Good luck with your system design interview! 📊 📈