Logging

Ashish Pratap Singh

3 min read

Debugging a single-server application is straightforward: you SSH into the machine and `tail` a log file.

But what happens when your application is a constellation of microservices, running on hundreds of ephemeral containers that can be created and destroyed in minutes?

This is why **logging** is so important. It enables:

- **Debugging:** Pinpointing the root cause of errors.
- **Monitoring:** Understanding system health and performance.
- **Auditing & Compliance:** Tracking user activity and meeting legal requirements.
- **Business Intelligence:** Analyzing user behavior and application trends.

---

1. What Is Logging?

Logs are timestamped records of events that occur within a system. They can be unstructured plain text or, preferably, structured data like JSON.

Typical Log Types:

- **Application Logs:** Events from your application's business logic (e.g., `User signed up`, `Payment failed`).
- **System Logs:** Events from the operating system or underlying infrastructure.
- **Access Logs:** Records of every request made to a server (e.g., from a web server or load balancer).
- **Security Logs:** Records of security-related events like login attempts or permission changes.

Unstructured Log Example:

```
[2025-10-05T14:01:00Z] ERROR: Payment failed for u
```

Structured (JSON) Log Example:

Json

```
{
  "timestamp": "2025-10-05T14:01:00Z",
  "level": "ERROR",
  "message": "Payment failed",
  "service": "payment-service",
  "user_id": 123,
  "trace_id": "abc-123-xyz-789",
  "details": {
    "reason": "Insufficient funds",
    "amount": 99.99
  }
}
```

JSON is easier to parse, filter, and aggregate.

Key Objectives of a Logging System

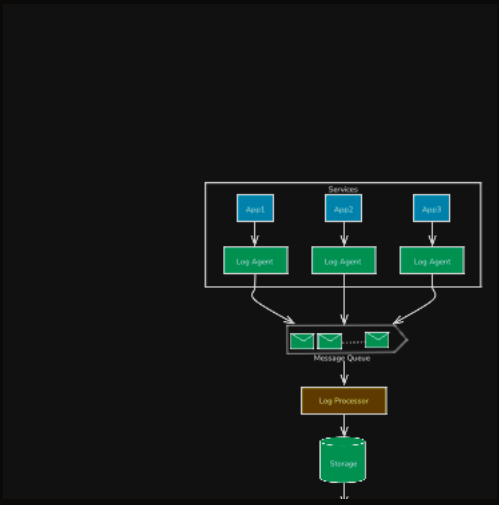
A robust logging system is designed to meet several key non-functional requirements.

- **Reliability**: don't lose logs during spikes or failures.
- **Scalability**: ingest millions of events/sec across regions.
- **Queryability**: fast search by time, service, level, fields.
- **Cost efficiency**: tiered storage (hot vs cold), compression, sampling.
- **Retention & compliance**: lifecycle policies, legal holds, GDPR/SOC2.

---

2. The Logging Pipeline

At a high level, a logging system is a data pipeline with several distinct stages.



Log Generation and Collection

This is the starting point of the pipeline, where logs are created and collected from their source.

- **How Logs are Generated:** Applications typically use logging libraries (like **Log4j** for Java, **Winston** for Node.js, or **Serilog** for .NET) to write logs to standard

output (`stdout`) or a local file.

- **Log Agents:** A lightweight agent, such as **Fluent Bit**, **Vector**, or **Logstash**, runs on each server or as a side-car container. Its job is to tail log files, collect logs from `stdout`, and forward them to the next stage.
- **Log Levels:** These are used to categorize the severity of a log message. Using them correctly is crucial to avoid drowning in noise.
    - **DEBUG:** Detailed information for developers.
    - **INFO:** Normal application behavior (e.g., service started).
    - **WARN:** Potentially harmful situations.
    - **ERROR:** Errors that impact functionality but don't crash the application.
    - **FATAL:** Severe errors that cause the application to terminate.

Log Aggregation and Transport

In a distributed system, logs are generated on many different machines. They need to be aggregated and transported to a central location.

- **Why Centralize?** Servers and containers are ephemeral. If a container crashes, its local logs are lost forever unless they've been shipped off the machine.
- **The Role of a Message Queue:** A message queue like **Apache Kafka** or **AWS Kinesis** acts as a central, durable buffer. Log agents send logs here. This decouples the log producers from the consumers (processors), absorbs traffic spikes, and prevents data loss if the downstream processing layer is slow or unavailable.

Log Processing and Indexing

Raw log data is rarely in the perfect format. The processing stage is where the magic happens.

- **Parsing:** Converting unstructured text logs into a structured format like JSON.
- **Filtering:** Dropping noisy or unnecessary logs (e.g., debug logs from production).
- **Enrichment:** Adding valuable context to logs. For example, adding geolocation data based on an IP address or user details based on a user ID.
- **Indexing:** The processed logs are sent to a search database like Elasticsearch, which builds an index to make them searchable. The index is like the index at the back of a book—it allows you to find information quickly without reading the whole book.

Log Storage

Once processed, logs need to be stored. The choice of storage depends on the access patterns and cost constraints.

- **Search Databases (Hot Storage):** Systems like **Elasticsearch**, **OpenSearch**, and **Grafana Loki** are optimized for fast text search and analysis. They are expensive but provide near real-time query capabilities. This is where you store logs for recent events (e.g., the last 7-30 days).
- **Object Stores (Cold Storage):** Services like **Amazon S3** or **Google Cloud Storage** are extremely cheap and durable, making them ideal for long-term archival of logs to meet compliance requirements.
- **Log Lifecycle Management:** A common pattern is to automatically move logs from hot storage to cold storage after a certain period (e.g., 30 days) and then delete them entirely after the full retention period (e.g., 1 year).

Search and Visualization

This is the user-facing part of the system, where engineers and analysts interact with the log data.

The most popular stack is **Kibana** (for Elasticsearch) or **Grafana** (for various data sources, including Loki). Other commercial tools include **Datadog**, **Splunk**, and **CloudWatch Logs Insights**.

Use Cases:

- **Debugging:** Searching for all logs with a specific `trace_id` to follow a request's journey through multiple microservices.
- **Monitoring:** Creating dashboards that visualize the rate of errors per service.
- **Security Audits:** Alerting on suspicious activity, like multiple failed login attempts from a single IP address.

---

3. Challenges and Considerations

Scalability Challenges

- **High Ingestion Rate:** Popular applications can generate terabytes of logs and millions of events per second. The entire pipeline, from the message queue to the storage layer, must be able to handle this load.
- **Indexing and Query Latency:** As the amount of data grows, indexing can slow down, and queries can become sluggish. Search databases like Elasticsearch use **sharding** (partitioning the index across multiple nodes) to scale horizontally.

- **Storage Cost:** Storing vast amounts of log data, especially in expensive hot storage, can become a significant operational cost.

Reliability and Fault Tolerance

A logging system that loses logs is not very useful.

- **"At-least-once" Delivery:** The system must be designed to ensure that logs are not dropped. This is achieved through acknowledgement and retry mechanisms at each stage.
- **Backpressure and Buffering:** If a downstream component (like Elasticsearch) is slow, the upstream components (like the log processor or message queue) must be able to buffer the data to prevent it from being lost.
- **Replication:** All components, especially the message queue and storage layers, should be replicated across multiple machines or availability zones to prevent a single point of failure.

Security and Privacy

Logs can contain sensitive information, so security is paramount.

- **Protecting Sensitive Data:** Personally Identifiable Information (PII) like emails, passwords, and credit card numbers should be **masked** or redacted from logs before they are stored.
- **Access Control:** The logging system should have strong access control policies to ensure that only authorized personnel can view logs.
- **Compliance:** For regulations like GDPR and HIPAA, you must have clear data retention policies and be able to delete a user's data upon request.

---

4. Best Practices

Here are some best practices to get the most out of your logging system:

- **Use Structured (JSON) Logs:** They are machine-readable and eliminate the need for complex parsing rules.
- **Set Appropriate Log Levels:** Use log levels (e.g., DEBUG, INFO, WARN, ERROR) to control the verbosity of logs and filter out noise from critical events.
- **Include Contextual Information:** Enrich logs with metadata such as timestamps, request IDs, user IDs, and service names to correlate events across systems. In a microservices architecture, include a unique `trace_id` in every log message to track a single request across multiple services.
- **Centralize Your Logging:** Don't let logs sit on individ-

ual servers.

- **Apply Log Sampling:** For extremely high-volume, non-critical logs (like CDN access logs), you can sample a percentage of them to reduce costs.
- **Log What Matters:** Don't log everything. Log key business events, errors, and warnings. Your code should tell you *how* it works; your logs should tell you *what* it's doing.
- **Implement Log Rotation and Retention Policies:** Manage disk space and ensure compliance by automatically rotating logs and archiving or deleting old logs.