## Logging

A log file records details of events occurring in a software application. The details may consist of microservices, transactions, service actions, or anything helpful to debug the flow of an event in the system. Logging is crucial to monitor the application’s flow.

Need for logging

Logging is essential in understanding the flow of an event in a distributed system. It seems like a tedious task, but upon facing a failure or a security breach, logging helps pinpoint when and how the system failed or was compromised. It can also aid in finding out the root cause of the failure or breach. It decreases the meantime to repair a system.

Logging allows us to understand our code, locate unforeseen errors, fix the identified errors, and visualize the application’s performance. This way, we are aware of how production works, and we know how processes are running in the system.

## Introduction of distributed Logging

In microservice architecture, logs of each microservice are accumulated in the respective machine. If we want to know about a certain event that was processed by several microservices, it is difficult to go into every node, figure out the flow, and view error messages. But, it becomes handy if we can trace the log for any particular flow from end to end.

The number of logs increases over time. At a time, perhaps hundreds of concurrent messages need to be logged. But the question is, are they all important enough to be logged? To solve this, logs have to be structured. We need to decide what to log into the system on the application or logging level

## Use Sampling

For example, there are people commenting on a post, where Person X commented on Person Y’s post, then Person Z commented on Person Y’s post, and so on. Instead of logging all the information, we can use a sampler service that only logs a smaller set of messages from a larger chunk. This way, we can decide on the most important messages to be logged.

What is a scenario where the sampling approach will not work?

Hide Answer
Let’s consider an application that processes a financial ATM transaction. It runs various services like fraud detection, expiration time checking, card validation, and many more. If we start to miss out logging of any service, we cannot identify an end-to-end flow that affects the debugging in case an error occurs. Using sampling, in this case, is not ideal and results in the loss of useful data.

The following severity levels are commonly used in logging:

DEBUG
INFO
WARNING
ERROR
FATAL/CRITICAL

Structure the logs
Applications have the liberty to choose the structure of their log data. For example, an application is free to write to log as binary or text data, but it is often helpful to enforce some structure on the logs.

We should consider the following few points while logging:

Avoid logging personally identifiable information (PII), such as names, addresses, emails, and so on.
Avoid logging sensitive information like credit card numbers, passwords, and so on.
Avoid excessive information. Logging all information is unnecessary. It only takes up more space and affects performance.
Logging, being an I/O-heavy operation, has its performance penalties.
The logging mechanism should be secure and not vulnerable because logs contain the application’s flow, and an insecure logging mechanism is vulnerable to hackers.

# Design of a distributed Logging Service

The functional requirements of our system are as follows:

Writing logs: The services of the distributed system must be able to write into the logging system.

Searchable logs: It should be effortless for a system to find logs. Similarly, the application's flow from end-to-end should also be effortless.

Storing logging: The logs should reside in distributed storage for easy access.

Centralized logging visualizer: The system should provide a unified view of globally separated services.

The non-functional requirements of our system are as follows:

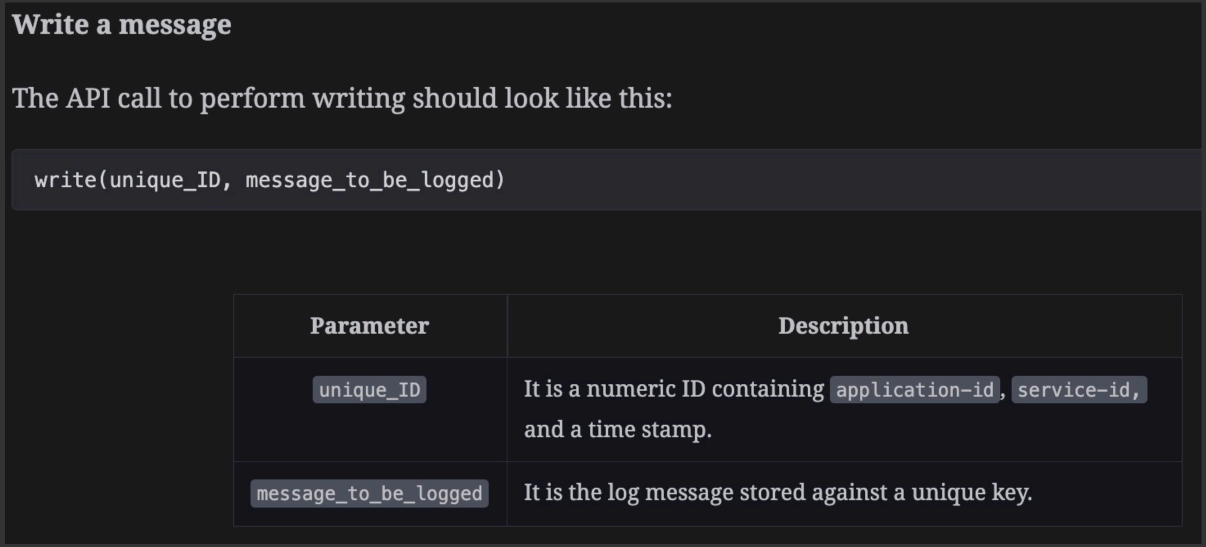
Low latency: Logging is an I/O-intensive operation that is often much slower than CPU operations. We need to design the system so that logging is not on an application's critical path.

Scalability: We want our logging system to be scalable. It should be able to handle the increasing amounts of logs over time and a growing number of concurrent users.

Availability: The logging system should be highly available to log the data.

Pub-sub system: We'll use a pub-sub- system to handle the huge size of logs.
Distributed search: We'll use distributed search to query the logs efficiently.

# API design

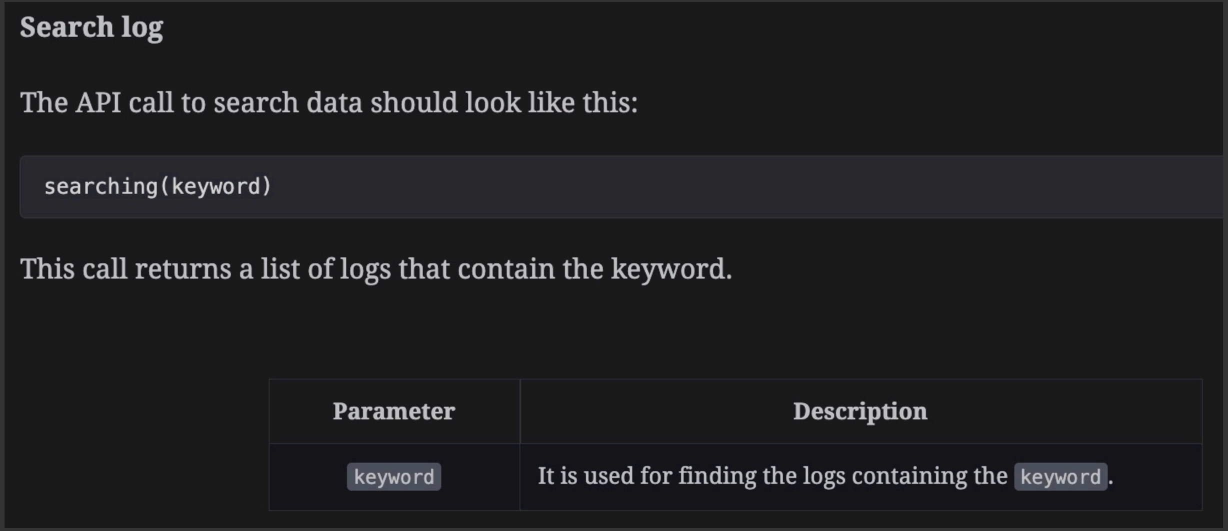


### Write a message

The API call to perform writing should look like this:

```
write(unique_ID, message_to_be_logged)
```

| Parameter | Description |
|---|---|
| `unique_ID` | It is a numeric ID containing `application-id`, `service-id,` and a time stamp. |
| `message_to_be_logged` | It is the log message stored against a unique key. |



### Search log

The API call to search data should look like this:

```
searching(keyword)
```

This call returns a list of logs that contain the keyword.

| Parameter | Description |
|---|---|
| `keyword` | It is used for finding the logs containing the `keyword`. |

# Initial Design

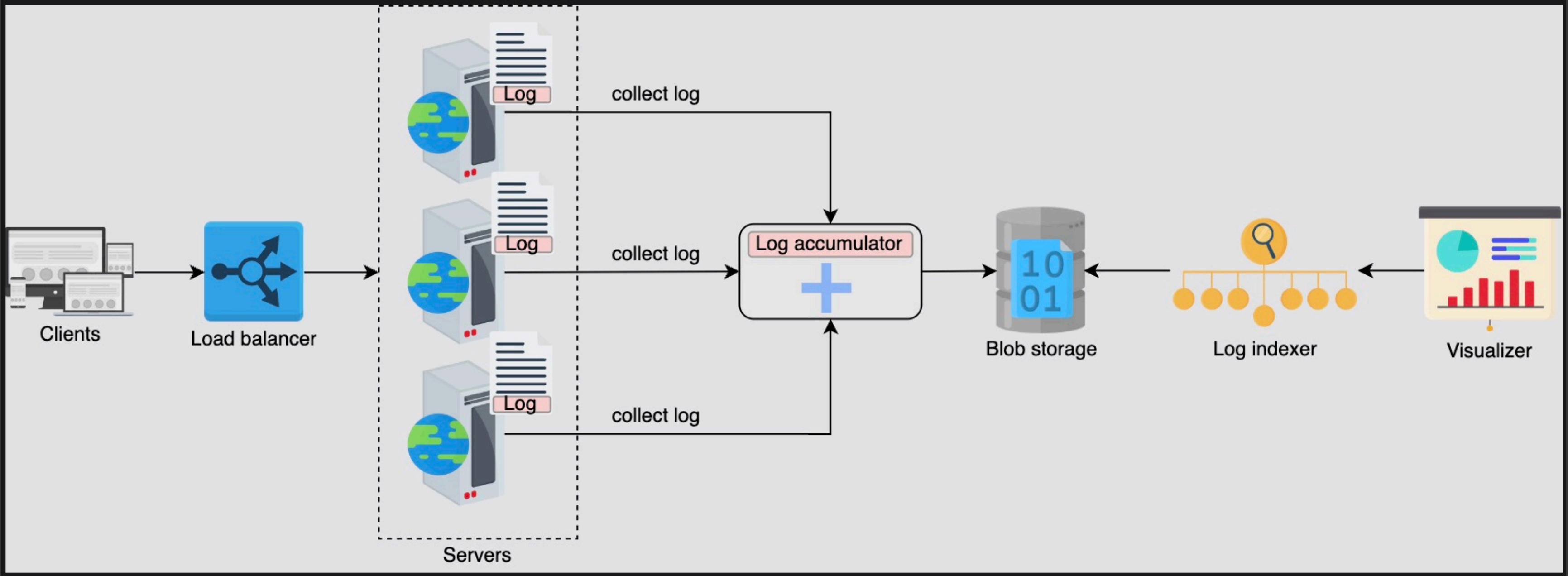
In addition to the building blocks, let’s list the major components of our system:

Log accumulator: An agent that collects logs from each node and dumps them into storage. So, if we want to know about a particular event, we don’t need to visit each node, and we can fetch them from our storage.

Storage: The logs need to be stored somewhere after accumulation. We’ll choose blob storage to save our logs.

Log indexer: The growing number of log files affects the searching ability. The log indexer will use the distributed search to search efficiently.

Visualizer: The visualizer is used to provide a unified view of all the logs.



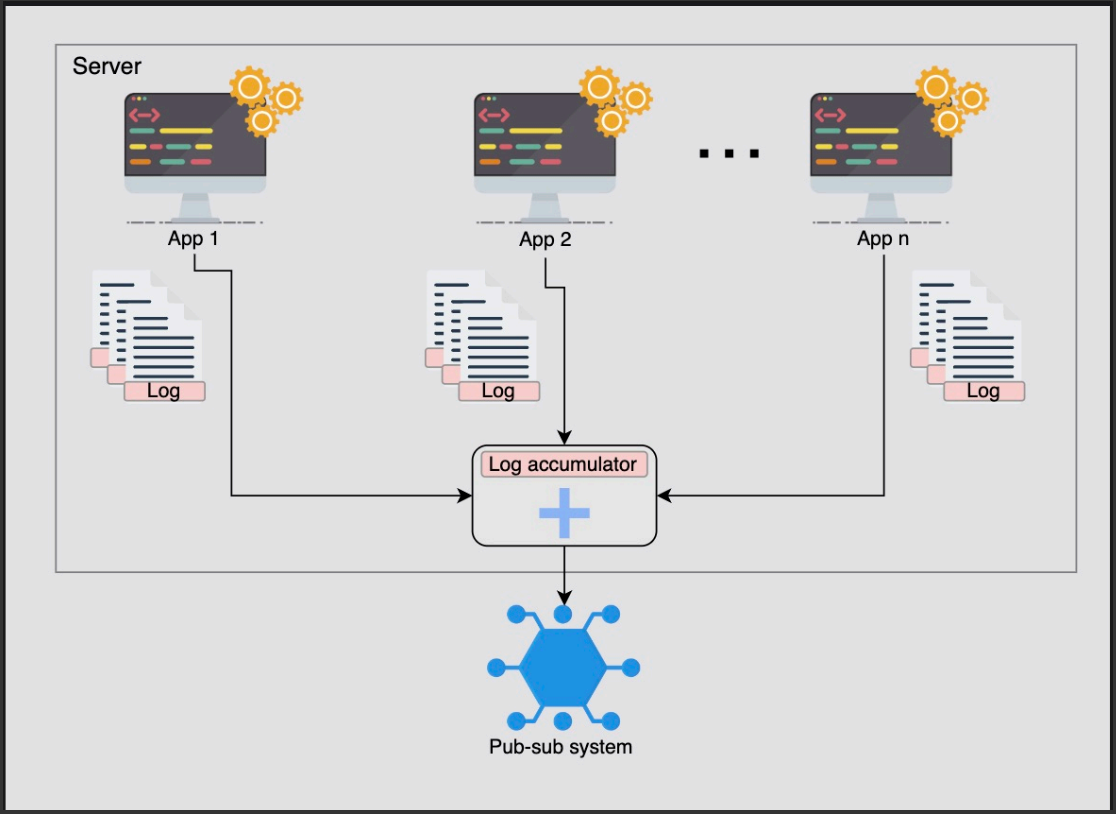
# Logging at various levels

## → In a server

Let's consider a situation where we have multiple different applications on a server, such as App 1, App 2, and so on. Each application has various microservices running as well. For example, an e-commerce application can have services like authenticating users, fetching carts, and more running at the same time. Every service produces logs. We use an ID with application-id, service-id, and its time stamp to uniquely identify various services of multiple applications. Time stamps can help us to determine the causality of events.

Each service will push its data to the log accumulator service. It is responsible for these actions:

Receiving the logs.
Storing the logs locally.
Pushing the logs to a pub-sub system.



How does logging change when we host our service on a multi-tenant cloud (like AWS) versus when an organization has exclusive control of the infrastructure (like Facebook), specifically in terms of logs?

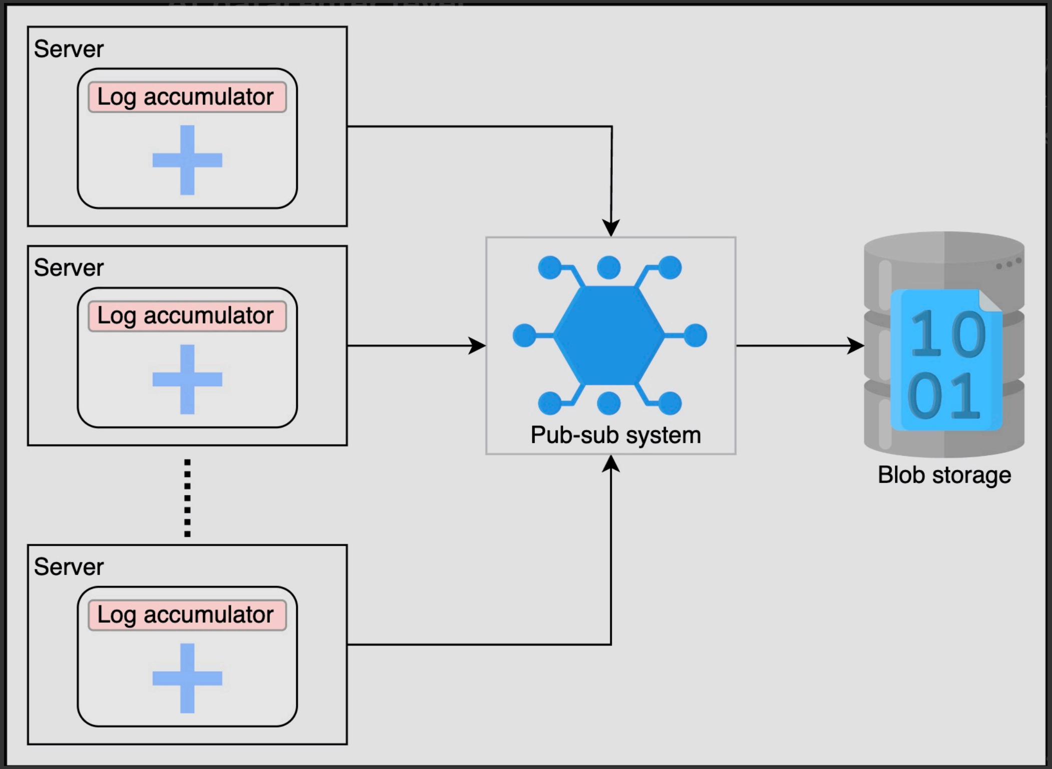
Hide Answer

Security might be one aspect that differs between multi-tenant and single-tenant settings. When we encrypt all logs and secure a logging service end-to-end, it does not come free, and has performance penalties. Additionally, strict separation of logs is required for a multi-tenant setting, while we can improve the storage and processing utilization for a single-tenant setting.

Let's take the example of Meta's Facebook. They have millions of machines that generate logs, and the size of the logs can be several petabytes per hour. So, each machine pushes its logs to a pub-sub system named Scribe. Scribe retains data for a few days and various other systems process the information residing in the Scribe. They store the logs in distributed storage also. Managing the logs can be application-specific.

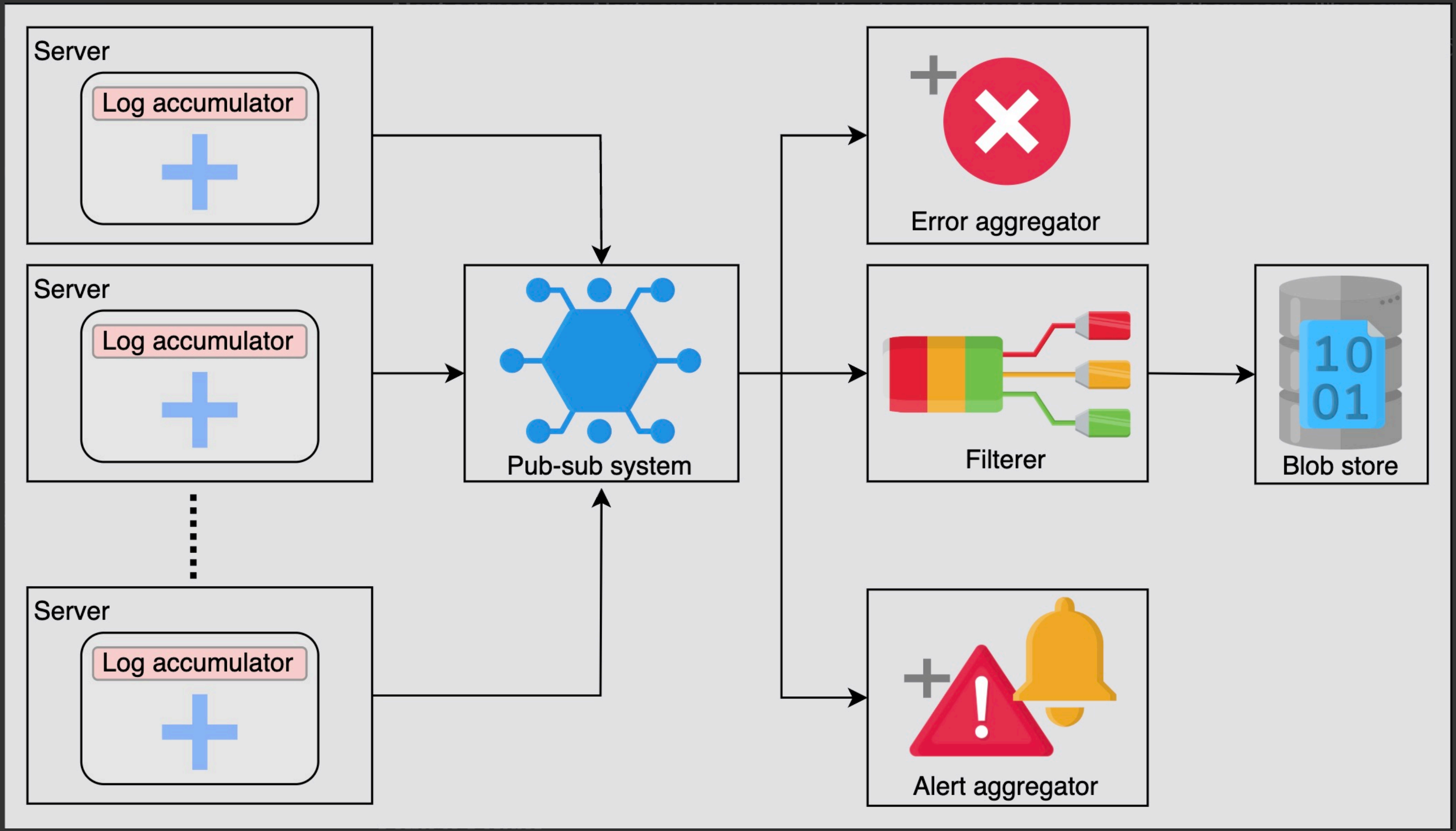
On the other hand, for multi-tenancy, we need a separate instance of pub-sub per tenant (or per application) for strict separation of logs.

At datacenter level



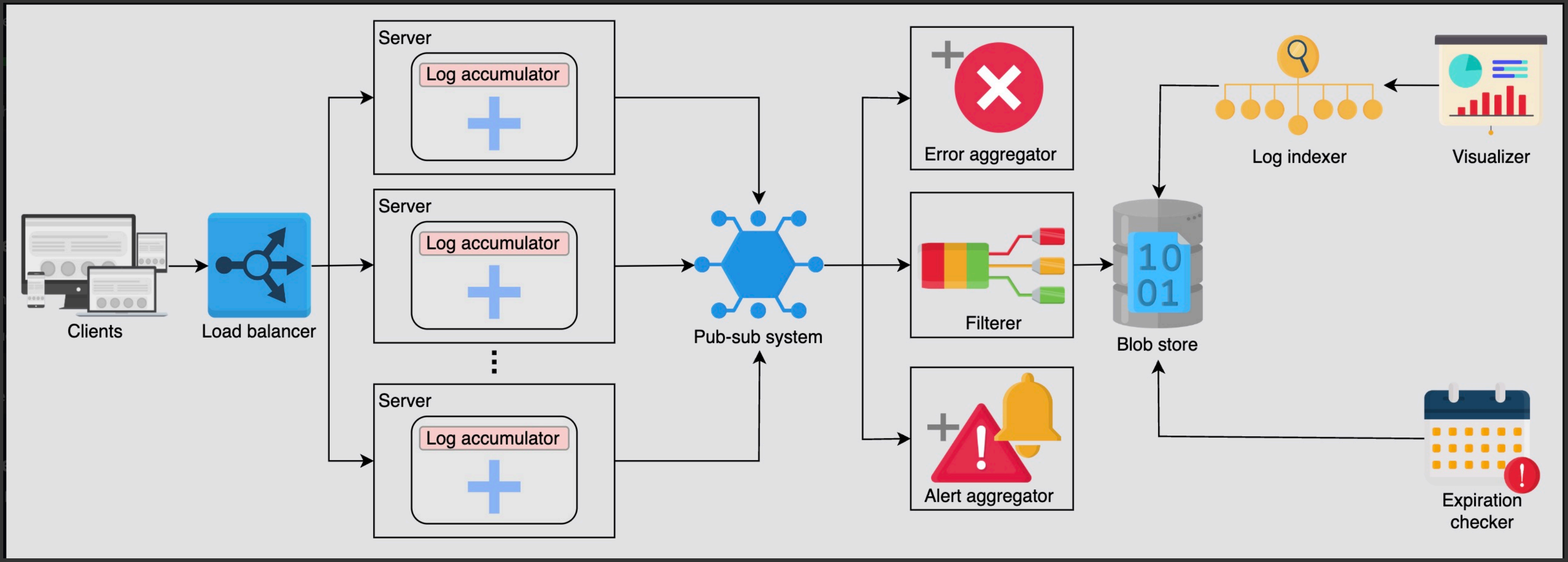
all servers in a data center push the logs in pub-sub system.

We use a horizontally-scalable pub-sub system.



Do we store the logs for a lifetime?

Hide Answer
Logs also have an expiration date. We can delete regular logs after a few days or months. Compliance logs are usually stored for up to three to five years. It depends on the requirements of the application.



We learned earlier that a simple user-level API call to a large service might involve hundreds of internal microservices and thousands of nodes. How can we stitch together logs end-to-end for one request with causality intact?

Hide Answer
Most complex services use a front-end server to handle an end user's request. On reception of a request, the front-end server can get a unique identifier using a sequencer. This unique identifier will be appended to all the fanned-out services. Each log message generated anywhere in the system also emits the unique identifier.

Later, we can filter the log (or preprocess it) based on the unique identifiers. At this step, we are able to collect all the logs across microservices against a unique request. In the Sequencer building block, we discussed that we can get unique identifiers that maintain happens-before causality. Such an identifier has the property that if ID 1 is less than ID 2, then ID 1 represents a time that occurred before ID 2. Now, each log item can use a time- stamp, and we can sort log entries for a specific request in ascending order.

Correctly ordering the log in a chronological (or causal) order simplifies log analyses.