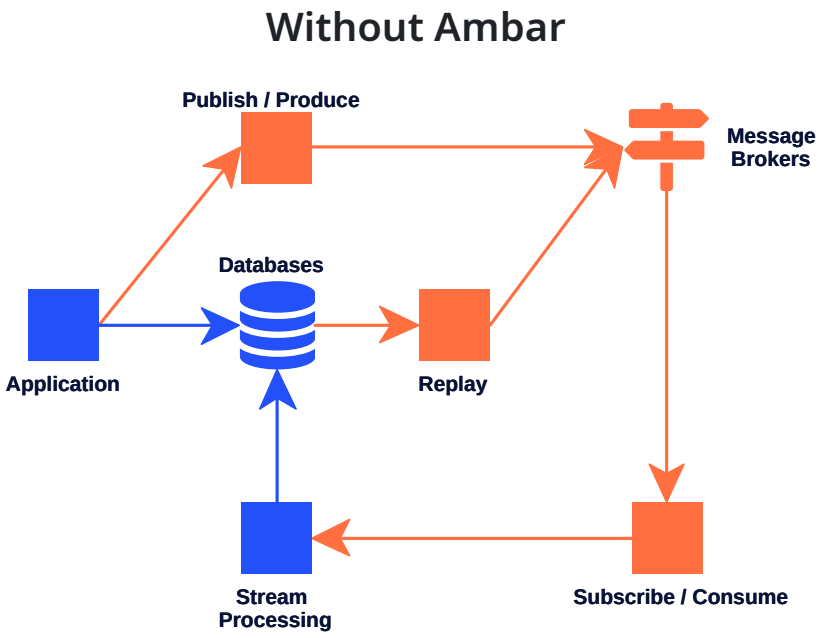
Ambar

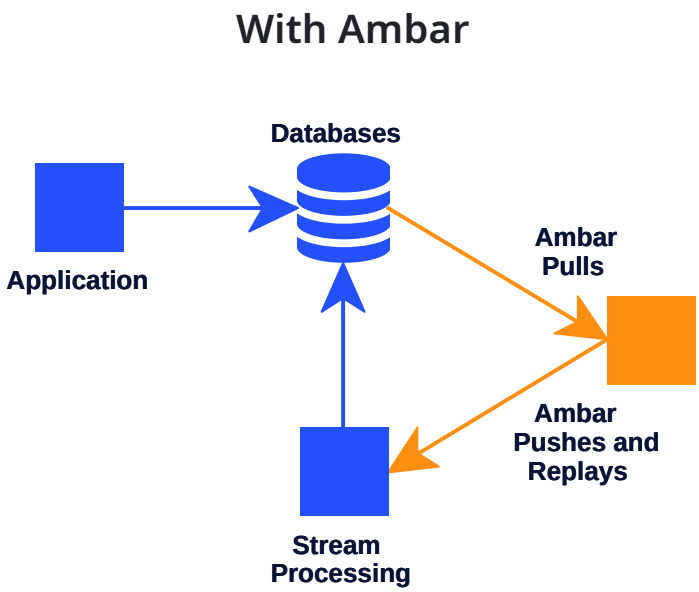
About Ambar

Ambar simplifies Data Streaming with a truly fully managed service that encapsulates ingestion, processing, and distribution, reducing the need for extensive code management and infrastructure maintenance. Using Ambar is easy: dispatch messages to your database and receive them via HTTP.

Ambar gives you a pub-sub model with filter, merge, and replay capabilities. It ensures exactly once, in-order, partitioned, ACID-compliant streaming. Moreover, you get horizontal scalability, high availability, and high durability.

How does it work? Ambar pulls data from append-only tables in your databases, which we call Data Sources. Ambar stores this data in distributed brokers that offer redundancy, durability, and availability while maintaining low latencies. From there, Ambar pushes your data to HTTP endpoints, which we call Data Destinations, in order, in parallel, with smart error handling and user-defined filtering. In other words, Ambar gives you Data Streaming without maintaining message brokers, background workers, producers, and consumers. **And it's dead easy. You can get started with 14 lines of configuration.**





Using Ambar

Data Sources

Create Data Sources using our web console, Terraform, Pulumi, or calling our API. Ambar will connect to each Data Source, capture data in real-time, and store it for processing. And voila! Ambar has now taken over the role of your *producers / publishers*. You only have to produce to a database (which you already know how to do).

For example, you have an existing MySQL table called `order_events` with the following columns:

```
auto_incrementing_id(BIGINT)
order_id(BINARY)
event_id(BINARY)
event_type(VARCHAR)
occurred_at(DATETIME)
event_payload(JSON)
```

Using Terraform, let's create two Data Sources. Notice that you can even use JSON path syntax to retrieve from a JSON column (`event_payload -> '$.vendor_id'`).

```
data_sources = [
  {
    "data_source_name" : "order_events",
    "incrementing_key" : "auto_incrementing_id",
    "partitioning_key" : "order_id",
    "unique_key" : "event_id",
    "additional_keys": "event_type,occurred_at,event_payload",
```

```
    "database_connection": {###}
  },
  {
    "data_source_name" : "order_events_by_vendor",
    "incrementing_key" : "auto_incrementing_id",
    "partitioning_key" : "event_payload -> '$.vendor_id'",
    "unique_key" : "event_id",
    "additional_keys": "order_id,event_type,occurred_at,event_payload",
    "database_connection": {###}
  }
]
```

Data Destinations

Create Data Destinations using our web console, Terraform, Pulumi, or calling our API. Ambar will send data to each Data Destination's HTTP endpoint. It isn't required, but you can apply filtering logic through the `filtering` configuration. And voila again! Ambar has now taken over the role of your *subscribers / consumers*. You only have to write HTTP endpoints (which you already know how to do).

Using Terraform, let's create a Data Destination that receives a subset of data from the Data Source `order_events`.

```
data_destinations = [
  {
    "data_destination_name": "stock_level_tracker",
    "data_destination_type": "json",
    "data_destination_http_endpoint": {"endpoint": "https://example.com/api/st
    "filtering": {
      "or": [
        {
          "and": [
            {
              "filter": "by_data_source",
              "match": {
                "operator": "=",
                "value": "order_events"
              }
            },
            {
              "filter": "by_key",
              "match": {
                "key_name": "event_name",
                "key_value": {
                  "operator": "in",
                  "value": [
                    "product_purchased",
                    "product_reserved",
                    "product_shipped"
```

```
                ]
              }
            }
          }
        ]
      }
    ]
  }
]
```

The http endpoint will then receive requests as follows:

```
{
  "data_source_name": "order_events",
  "data_destination_name": "order_events_by_vendor",
  "payload": {
    "auto_incrementing_id": "my-event-name",
    "order_id": "MDUwMDAyNzAzMzMwMTAwMDM1QzRBODkxODAwMTA1QTA=",
    "event_id": "NEE4OTE4MDAxMDVBMDA1MDAwMjcwMzMzMDEwMDAzNUM=",
    "event_type": "product_reserved",
    "occurred_at": "2020-12-28 23:05:13",
    "event_payload": "{vendor_id: \"pen_vendor_541\", product_name: \"pen\"}",
    "event_payload -> '$.vendor_id'": "\"pen_vendor_541\""
  }
}
```

The endpoint can then answer with an acknowledgement

```
{
  "result": {
    "success": {}
  }
}
```

Or a negative acknowledgement,

```
{
  "result": {
    "error": {
      "policy": "must_retry",
      "class": "...",
      "description": "..."
    }
  }
}
```

In the case of a negative acknowledgement or an error, Ambar will retry the request with exponential backoff. Additionally, you can instruct Ambar to log the error and keep going by setting the negative acknowledgement's policy to `"keep_going"`.

Guarantees

Ambar provides incredible guarantees -- ACID HAD PIE. We're coining HAD and PIE.

ACID

A (Atomicity): Ingestion and checkpointing occurs atomically.

C (Consistency): Immediate availability of ingested records for Data Destination processing after committing Data Source records.

I (Isolation): Strict isolation of ingress and egress operations within each partition.

D (Durability): Retention of all data (with replication) in the event of a system shutdown, partial or total.

HAD

HA Highly available - 99.99% by default (99.999% with additional cost)

D Highly durable - 99.9999999%

PIE

P (Partitioned): Parallelized calls to destination HTTP endpoints based on the Data Source partitioning key.

I (In Order): Calls to destination HTTP endpoints in the order found in the Data Source, per partition.

E (Exactly Once): Destinations process each Data Source record exactly once.

Customer Requirements

To obtain Ambar's guarantees, we only require the following:

1. Data Sources must be append-only.
2. Data Sources must have an incrementing key, a unique key, and a partitioning key.
3. Data Destination endpoints must be idempotent (deduplicate messages across your unique key).

What's under the hood?

Infrastructure

We host Ambar infrastructure across AWS, GCP, and Azure in our cloud accounts. Of course, we hide away producers and consumers, but they're still under the hood. Unlike other vendors who are tied to Kafka Connect, Apache Flink, Debezium, and more; our producers and consumers are proprietary so we can deliver market-leading features and guarantees. And if you were wondering, we write our code in Rust.

Security

We place a strong emphasis on the security of our customers' data streams, so we implement stringent measures to safeguard data.

1. Ambar only supports authenticated connections to Data Sources and Destinations.
2. Ambar only supports connections with encryption in transit to both Data Sources and Data Destinations.
3. Ambar stores data with encryption at rest.
4. Internally, our passwords comply with NIST 800-53, and we require multi-factor authentication across our systems.
5. Ambar employs the principle of least access with staff. We only deploy with infrastructure as code and automated pipelines; thus we prevent internal access to customer data.
6. Ambar employs the principle of least access to infrastructure. For example, our producers and consumers run in isolated environments without access to unneeded OS functionality.
7. Ambar maintains automated tooling to keep our software and its dependencies up to date.
8. Ambar maintains automated reporting that alerts us of security issues at all layers, from our cloud vendor APIs to network flow logs.