Using an architectural pattern as an example, this article explains how to use Spanner (/spanner/docs/) as an event-ingestion system and Pub/Sub (/pubsub/docs/) as an event ledger to create a system that can do the following:

- Write to an event source that is highly available.
- Publish those writes as events for other systems to use.
- Archive events for playback.
- Load events into a system for analytics.
- Filter events into a system for fast querying.

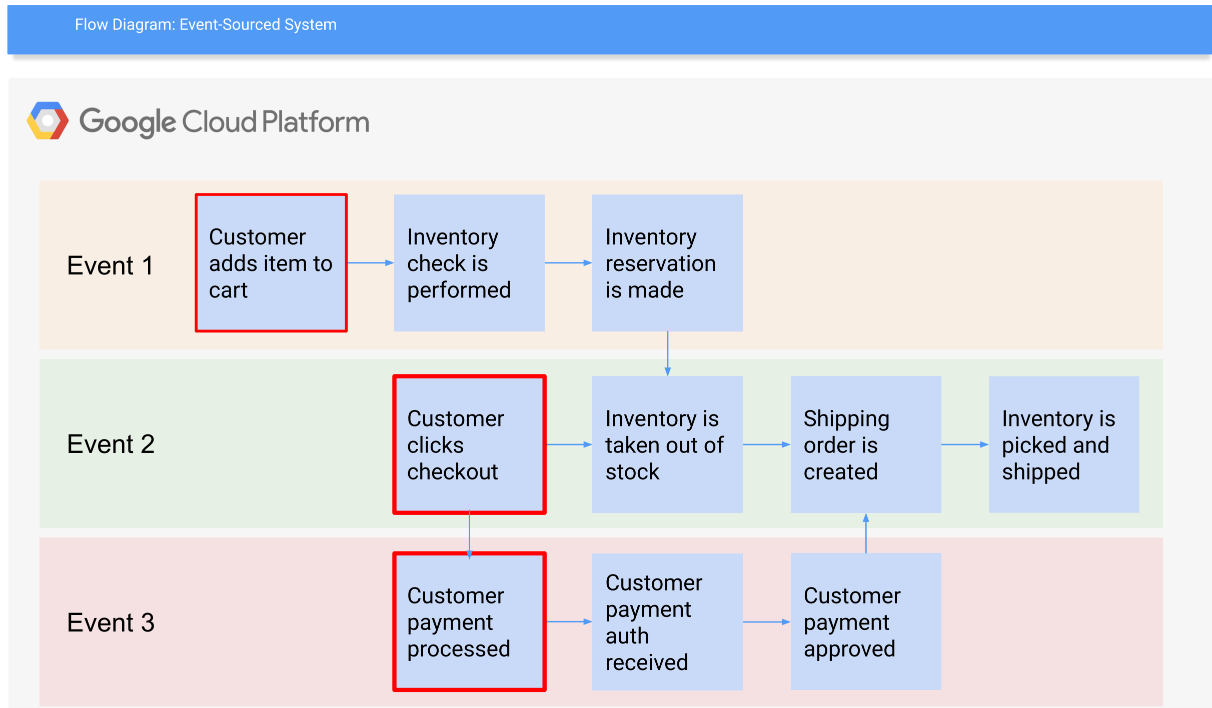
This article is designed for software engineers interested in learning about the uses, trade-offs, and components of an event-sourced system (https://serverless.com/blog/rob-gruhl-serverless-event-sourced-nordstrom-emit-2017/). Learn how to create a number of apps and services by using Cloud Functions (/functions/docs/) to support your event-sourced architecture.

You can use this architectural pattern anytime you need to take an action based on writing new data to a data source, in this case, Spanner.

This pattern is useful for managing the following scenarios:

- Ecommerce shopping carts
- Order management and supply chain
- Wallets, payments, and charge resolutions

The following diagram illustrates the flow of an ecommerce shopping cart system.



In complex systems such as ecommerce and payments, it's useful to track transactions by using event-based architectures. For example, when a customer adds an item to a shopping cart or processes a credit card for payment, you might want to trigger several downstream processes to verify that the item is in stock and that there are funds in the custmer's account. You want to make sure that customers can place orders at any time because your business depends on it.

The following are examples of design criteria that suggest using an event-sourced system arhitecture::

- Must handle the writing of orders and payments to the system with high availability.
- Must verify that your customers receive the items they ordered (and only the items they ordered), and that they were charged the correct amount for the purchase.
- Must have deterministic failure modes (that is, you are certain that your write failed or succeeded).
- Must include a mechanism to notify dependent services that a system write that they were interested in was made.

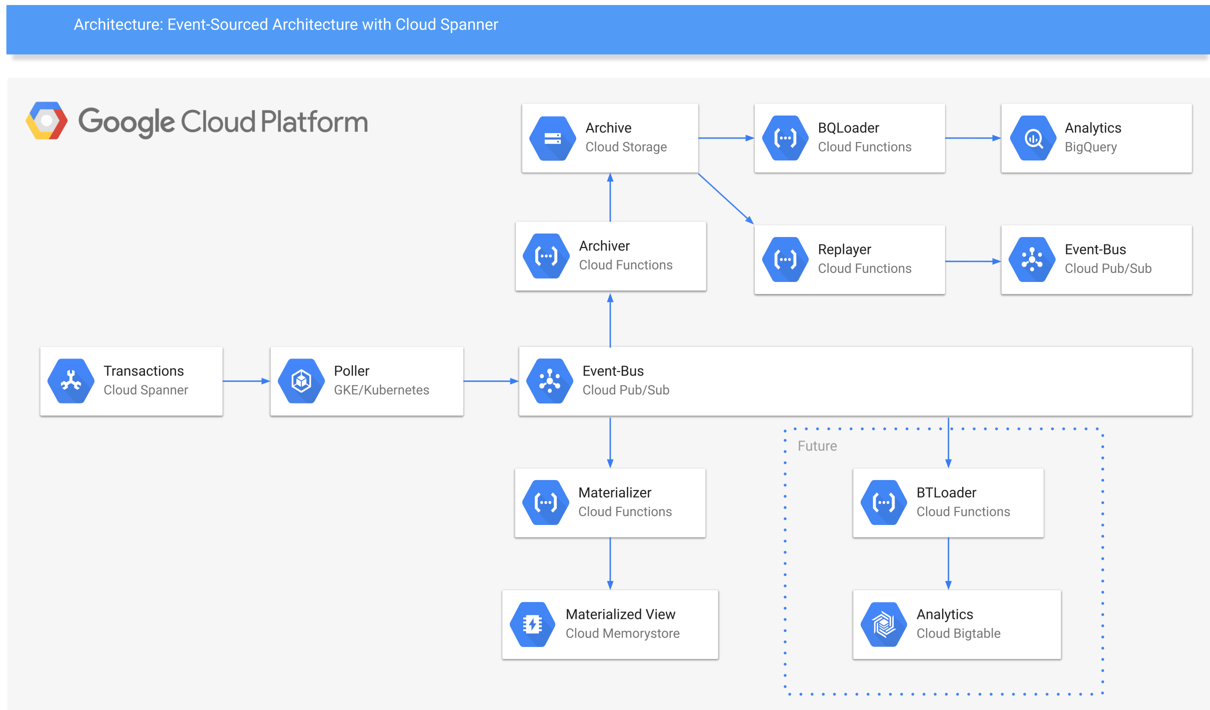
This article describes a system that can satisfy all these requirements and provide flexibility for adding functionality later.

First you need a service that can accept writes with high availability. That system must also provide deterministic failure modes. The system must know when a write fails so that your writer can retry the write without fear of duplicating all or a portion of that write.

The canonical application for this scenario is a database that supports atomicity, consistency, isolation, and durability (ACID) transactions (https://wikipedia.org/wiki/ACID_(computer_science)), but making databases highly available, especially for writes, is difficult. Replication can cause inconsistencies in data and can add cost and complexity to your architecture. Compounding complexity is the largest design risk when high availability is a priority.

Additionally, typical high-availability database configurations (/sql/docs/mysql/high-availability) cannot handle failures across availability zones without incurring replication delays. As those delays increase, so does the probability that additional failures might lead to the loss of all of the writes that are in transit to the replica node in another availability zone. These additional failures mean that the writes aren't fully written to the replica prior to the failure in that zone.

The following diagram illustrates an event-sourced architecture with Spanner designed to address the issues of complexity and cost associated with traditional high-availability databases.



This event-sourced architecture relies on the following components, which you use Cloud Functions to create. These components consist of apps and Cloud Functions.

- Poller app: Polls Spanner, converts the record format to Avro (https://wikipedia.org/wiki/Apache_Avro), and publishes to Pub/Sub.
- `archiver`: Gets events triggered by messages published to a Pub/Sub topic and writes those records to a global archive in Cloud Storage.
- `bqloader`: Gets triggered by records written to Cloud Storage and loads those records to a corresponding BigQuery table.
- `janitor`: Reads all entries written to the global archive at a fixed rate and then compresses them for long-term storage.
- `replayer`: Reads the records in order from long-term storage, decompresses them, and loads them into a new Pub/Sub stream.
- Materializer app: Filters records written to Pub/Sub, and then loads them to a corresponding Redis (materialized view) database for easy query access.

After you have a system in place to accept writes, your downstream services must be notified every time something is written to the system.

Traditional databases do this in a few different ways, but usually with some variation of listening to the write ahead log (WAL) (https://wikipedia.org/wiki/Write-ahead_logging) or the change data capture (CDC) (https://wikipedia.org/wiki/Change_data_capture) stream of a database. These solutions aren't in a useful format for readability. The format was designed to represent and stream the changes made to the record. It wasn't designed to inform a downstream system of events and pass along the relevant context about that event. A binary representation of just the changes, and not a complete record, isn't useful in most situations. Another downside of that format is that it isn't human readable, which makes debugging and auditing the stream extremely difficult.

Instead, you can create something that polls the database for all new entries and then passes them along to the downstream system. Polling services are a common way to process new records written to a database and have the following advantages:

- Simple to understand and write.
- Low overhead when written correctly.
- Stable and independent.
- Error-tolerant, both for queries and record parsing.
- Flexibility in how changes are filtered and represented.

Trade-offs to writing a polling service instead of using a conventional WAL or CDC include:

- Polling the database at a short interval (less than one second) can add load to your database.
- Depending on your table layout, the query you use to poll, and how many other apps are currently polling your database, a polling service could create resource contention (locks) with other apps.
- Polling might require you to use larger database machines and more expensive storage (such as SSD) to handle the additional load and resource contention.

You can help to mitigate some of these trade-offs in Spanner by using read-only transactions for your polling reads (/spanner/docs/reads) and make sure you are using SQL best practices for efficient and effective queries (/spanner/docs/sql-best-practices).

To find all new records for a given time period, you can use the commit timestamp (/spanner/docs/commit-timestamp) feature in Spanner. The commit timestamp is based on TrueTime (/spanner/docs/true-time-external-consistency) technology and gives Spanner a globally consistent representation of when a write was committed to the database. Spanner can accept writes in many regions around the world, and your poller can create a ledger that contains an accurate accounting of events in order.

Cloud Functions (/functions/) is an event-driven, serverless, computing platform on Google Cloud. These functions are stateless snippets of code that run in response to a trigger, such as an HTTP request or an event trigger (/functions/docs/concepts/events-triggers)). In the case of event-sources systems, Cloud Functions typically represent the individual tasks associated with an event being published. Because they are serverless, they scale with your request volume and don't require additional operational intervention.

Some trade-offs with Cloud Functions are:

- Response times can be inconsistent.
- Logging and tracing can be more difficult based on the ephemeral nature of the startup and execution.
- Must be stateless and idempotent as retry on failure is typically automatic.
- Can be difficult to debug and reproduce errors locally depending on the state of your logging environment.

A *ledger* is an append-only record of events published on an event bus or a message queue of some kind. You can subscribe or listen for an event by either subscribing to a particular event bus or a message queue topic, or by filtering the collection of all messages by the noun, verb, or particular metadata you're interested in.

In this pattern, the ledger is a single Pub/Sub topic. You can use the eventing system to trigger Cloud Functions every time a record is written to the stream.

Make sure your subscribers don't acknowledge (/pubsub/docs/subscriber) the message, so it remains available for other interested functions. Also, there is a limited message retention

window (/pubsub/docs/subscriber) of messages on your queue, so you must create a Cloud Function that backs these messages up in an archive. When you want to play back archived messages, you can use a Cloud Function to read archived messages and publish them to a new Pub/Sub topic for consumption.

The job of the poller is to query the database at a fixed-time interval and ask for all records that occur after a particular point in time, sorted by the commit timestamp, and in ascending order (oldest record first).

This design means that you must keep track of the last timestamp that the poller saw, and you must bootstrap the system for the first run. You can keep track of this timestamp by storing it in application state, writing it to another database, or asking the ledger for its most recent record and parsing the timestamp from there.

Storing the record in another database might add some additional complexity, because it creates a dependency on another system which adds another point of failure. It's better to keep the last process's timestamp in local app storage and only resort to querying the ledger if the app fails or is restarted for some reason and the internal state is lost.

The requirements for the poller are as follows:

- Fixed and consistent polling interval.
- Polling interval is less than 1 second.
- Bootstraps the system the first time it's run.
- Queries all records occurring after the previously recorded timestamp.
- Serializes each record into an individual Avro record.
- Publishes each Avro record to a Pub/Sub-based event ledger.
- Sleeps until the duration of its polling interval has elapsed.
- If failure occurs, automatically retries within its fixed-time interval.

- If failure occurs, when the poller restarts, it queries the Pub/Sub stream to obtain the last recorded timestamp. If that fails, you can start the poller with a manually configured timestamp. You can also obtain the timestamp from the Cloud Storage archives.

Next, you design the poller. There are at least three different ways to design it, each with their own trade-offs.

You can use Cloud Scheduler to schedule a Cloud Function to poll the database, you can launch your poller in a Kubernetes cluster as a cron job and schedule it at the interval that you want, or you can have a service which continuously runs in a Kubernetes pod and polls the database at a fixed delay, maintaining and updating the last processed timestamp in memory.

Knowing that you must query the database at a fixed-time interval, you might consider using Cloud Scheduler to schedule a Cloud Function to poll the database (/scheduler/docs/tut-pub-sub) and then send the new records to a Pub/Sub stream.

This approach works, but it does involve two trade-offs. First, you must figure out a way to preserve the application state because Cloud Functions are stateless by definition. Preserving the application state involves introducing an additional database, and adds some inconsistencies in latency between events being written and events being added to the ledger. Cloud Functions can sometimes take longer to spawn and run than expected. This delay becomes more pronounced the shorter your poll interval becomes.

If all of your foreseeable downstream consumers can tolerate a variable latency that might exceed one second on occasion, a scheduled Cloud Function might be the best option for your system design. In that case, your Cloud Function tracks the last timestamp processed by querying the Pub/Sub stream, or maintains that state in Cloud Storage. While reducing the management complexity of something like Kubernetes, an additional trade-off to consider in the case of the Cloud Function is that those additional queries add latency due to the additional call for the timestamp, and thus can add an additional point of failure for your polling system. If you can tolerate that additional call latency, you can use a Cloud Function here.

Another option is to launch your poller in a Kubernetes cluster as a cron job (/kubernetes-engine/docs/how-to/cronjobs) and schedule it at the interval you want. Unfortunately, this approach has similar trade-offs as using Cloud Scheduler with the additional complexity of adding Kubernetes. However, you can launch the job as a service on Kubernetes and have it sleep for a set interval, which you control in the main-event loop of the app. You can maintain

state and have complete control over the polling latency and the retry semantics. While this choice comes with the additional complexity of Kubernetes, which you can mitigate by using Google Kubernetes Engine (GKE) (/kubernetes-engine/docs/), it provides you maximum control over the following:

- The state between polls (last timestamp).
- The latency between polls.
- Retry semantics and duration if the Spanner read or the Pub/Sub write fails.
- Automatic restart of the poller service if it dies or quits unexpectedly.

The first time you run the poller, it sets up all of the components required to run the event-sourced system. This includes the Pub/Sub stream with the correct name (ideally the name of the table) and the Cloud Storage bucket for the `archiver` to publish to. After all of the system components are set up, the bootstrap process of the poller performs an initial table scan, and processes all of the existing data. After the poller has finished, it moves to its fixed-polling interval and passes along the last processed commit timestamp for the polling system to use on its first processing run.

After you have the poller scheduled and it's pulling the latest data, you need to represent that data on the ledger. Because this is a generalized solution (that is, you can reuse this poller for many different tables with different schemas) writing table-specific pollers and ledger consumers isn't a scalable solution. You also need to be able to add different consumers to the ledger without them having to know the versioned schema variations beforehand.

The following are some potential use cases:

- Long-term archiving of all transactions.
- Loading transactions into a data warehouse for analytics.
- Loading transactions into a NoSQL database for feeding machine learning models and potentially caching the answers to frequently asked questions closer to the user.

For these use cases, consider using a serialization (https://wikipedia.org/wiki/Serialization) format, such as Avro, JSON, or Protobuf. BigQuery (/bigquery/), Google's data warehouse solution,

supports ingesting data directly from Avro files. Avro is the preferred format for loading data into BigQuery. Loading Avro files has the following advantages over JSON:

- Faster to load (/bigquery/docs/loading-data#loading_compressed_and_uncompressed_data). The data can be read in parallel, even if the data blocks are compressed (/bigquery/docs/loading-data-cloud-storage-avro).
- Doesn't require typing or serialization.
- Easier to parse because there aren't the inherent encoding issues sometimes found in other formats'.

When you load Avro files into BigQuery (/bigquery/docs/loading-data), the table schema is automatically inferred from the self-describing source data.

Protobuf is an alternative to Avro, but for this use case Avro has two distinct advantages:

- Direct ingestion support in BigQuery.
- Schema is contained in the data object.

The last advantage lets consumers of the ledger pull the data off and inspect it. They don't have to deserialize JSON and hope the format doesn't change, or pull a version of a schema registry for a given version of that object.

For those reasons, serialize your tables from Spanner into an Avro object for each transaction before you place it on the ledger. For more information, see how to transform a Spanner table to an Avro record. (https://github.com/GoogleCloudPlatform/DataflowTemplates/blob/master/src/main/java/com/google/cloud/teleport/spanner/ExportTransform.java)

After you decide on the deployment model and serialization format, consider your language options for writing the poller app. As the goal is to read data from Spanner and write to Pub/Sub, you are limited to the languages supported by the Google Cloud APIs. Those languages are C#, Go, Java, Node.js, PHP, Python, and Ruby.

Of the languages Spanner currently supports (/spanner/docs/reference/libraries), Avro officially supports (https://avro.apache.org/docs/1.8.2/) C#, Java, Python, PHP, and Ruby. Because you want to have as much control as possible over the latency of your app, and you want to process the queried tables in multiple threads, Java is a good option.

The poller is the main app in the service, but there are several other consumers of the stream.The first app you need is something that subscribes to the stream and archives each message into Cloud Storage for historical reference. This system stores each record as a separate file in a bucket with the same name as the table.

Every hour (or based on your transaction frequency) there is another service that takes those individual transaction records and compresses them into a larger file. Depending on the frequency of your transactions and the size of your transaction records, you might consider compressing the individual transaction Avro files as well.

After you have all of your historical transactions archived in Cloud Storage, you can:

- Directly populate BigQuery with transaction data for analytics.
- Playback historical records for training or testing of other systems.
- Rebuild the content of the system of record (the Spanner database in this case) if it gets lost or corrupted.
- Create a historical read-only replica for reporting or audit.
- Create a version of the database for staging or test environments.

Create a Cloud Function called `archiver` that is triggered anytime a transaction is added to the stream. You need to create a new function and trigger per topic (a table in Spanner). Every time a transaction is added to the configured Pub/Sub topic, the `archiver` Cloud Function grabs the Avro record and writes it to a Cloud Storage bucket with the same name as the table. The Cloud Function names the file with the table name, captures the date and time down to the millisecond, plus 4 random digits. This naming scheme creates an ID similar to a universally unique identifier (UUID) (https://wikipedia.org/wiki/Universally_unique_identifier).

Now you can create a new Cloud Function called `bqloader` to load that Avro file directly into BigQuery. The function is triggered when `archiver` uploads the file to Cloud Storage. Every time a transaction is loaded into Cloud Storage this Cloud Function appends that transaction entry

into the correct BigQuery table. If you want to further reduce latency for things like analytics or feeding machine learning models, you can stream (/bigquery/streaming-data-into-bigquery#bigquery_table_insert_rows-python) your data into BigQuery one record at a time by using the `tabledata.insertAll` (/bigquery/docs/reference/v2/tabledata/insertAll) method. This approach enables querying data without the delay of running a load job. Keep the quotas (/bigquery/quotas#streaming_inserts) for loading records into BigQuery in mind.

The `archiver` Cloud Function writes the Avro file to a Cloud Storage bucket. Next, you create another Cloud Function called `janitor` to compress all of the individual transactions in a single compressed file for long-term storage. `janitor` names this newly created file by using the table name, the date, and the time span included in the files. For example, if `janitor` is scheduled to run every hour, the file name might be `table1-jan_1_2019_1200-1300.tar.gz`. The `janitor` Cloud Function isn't a requirement, but helps to keep the storage costs down and the Cloud Storage buckets organized.

If you want to play the archived Avro files in Cloud Storage, you need another Cloud Function, called `replayer`, that is triggered by HTTP. The `replayer` Cloud Function takes the archived files for the time period you requested to be replayed, expands them, and publishes them, in order, on a new Pub/Sub stream.

This Cloud Function is triggered by using an HTTP `POST` (/functions/docs/calling/http) request that supplies the time period that you want replayed. The Cloud Function responds with the name of the Pub/Sub stream after it was finished, or with an error code and description if it couldn't properly load all of the archived data on to the new stream.

If your archived files become too large to consistently replay or replay becomes too slow for your use case, this app might require something more sophisticated. You can break your archives into smaller sections or make a different language selection for your `replayer` Cloud Function.

Another option is a Dataflow job, to break the job into smaller tasks and run in parallel. Implementing such a system is beyond the scope of this document, but there are great examples located in the Google Cloud GitHub repository

(https://github.com/GoogleCloudPlatform/DataflowTemplates/) and the Dataflow documentation (/dataflow/docs/).

Having these events on the ledger, each representing a single transaction, you can integrate different kinds of services into the system seamlessly without having to write any additional code and without having to coordinate between different app teams.

For example, this app listens for all messages on a certain topic, filters by a client's name, and creates a materialized view of that customer's relevant data. Ideally, you can use this app when you need to query information about a user and you need access to that information with very low latency.

If you can analyze the data early, put them in fast storage, and return a detailed response, you can improve performance for your frequently run queries.

This materializer app can consist of a Cloud Function that is triggered by a Pub/Sub topic, filters the information by client ID and, if the client ID matches the one it is interested in, the Cloud Function writes the relevant data to Memorystore (/memorystore/docs/redis/).

Building an event-sourced system architecture can create new functionality and add flexibility for any system that needs to react to or understand the relationship between events. When deterministic ordering or high-ingest throughput is important, using Spanner as an event-ingestion system and Pub/Sub as an event ledger can build a robust and reliable foundation for your event-sourced architecture. After you set up an event-sourced foundation, you can discover the many ways in which the problems that were once complicated to resolve can become simple Cloud Functions or Pub/Sub subscriber solutions.

- Functional poller, `archiver`, and `replayer` examples (https://github.com/GoogleCloudPlatform/spanner-event-exporter)

- Installing Istio on GKE (/istio/docs/istio-on-gke/installing)
- Triggering Cloud Functions from Pub/Sub (/functions/docs/concepts/events-triggers)
- Deploying Cloud Functions from Cloud Source Repositories (/source-repositories/docs/deploying-functions-from-source-repositories)
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).