Event-Driven Applications

Paving the Path to a Responsive and Reactive Real-Time Business

June 2023

US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com.

Table of Content

The rise of the humble event 3

How are event-driven applications different? 3

The evolution of event-driven technology 5

Fragmentation + friction = frustration 7

A Developer's Path to Productivity 8

Enabling Technology for Event-Driven Apps 9

Driving innovation with event-driven apps 13

1. Event-driven microservices 13

2. Event-driven streaming analytics 14

3. Event-driven Operational Data Layer (ODL) 15

4. Event-driven Extract Transform Load (ETL) 16

MongoDB in Event-Driven Apps Today 17

Getting started with event-driven apps 18

The rise of the humble event

The world moves in real time, and your apps need to do the same. Real-time applications bring digital experiences to life for your customers and speed time to insight for your business.

For customers, real-time apps stand out because they immediately react and respond to the constantly changing world around them. Think:

- Receiving hyper-personalized offers on an ecommerce site tailored to intent, channel, and location.
- Balance updates in a banking app as soon as debits and credits are processed.
- Instantly recalculating route plans to avoid traffic delays caused by a sudden breakdown on the road ahead.

At the same time, you need continuous and instant insights into your digital business. With those insights your systems unlock higher efficiency and profitability by taking intelligent decisions and action on live data in real time, rather than on aged and stale data. Think:

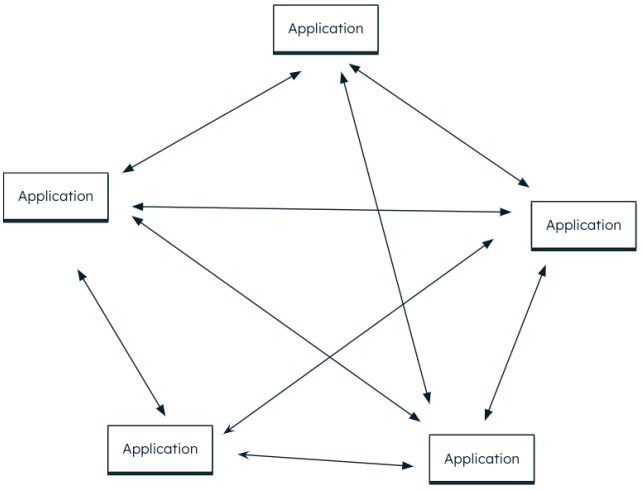
- Detecting and blocking fraudulent transactions during payment processing.
- Dynamically adjusting prices and inventory levels in response to volatile market demand.
- Analyzing sensor telemetry to detect and remediate potential equipment failures, avoiding costly outages.
- Extracting features from streaming data and feeding them into your machine learning models to make better predictions by using fresher data.

The question is how you can do all of this ahead of your competitors? The answer is what we call ***event-driven applications***.

With an event-driven approach, the constant flows of data exchanged between your business processes are continuously captured and actioned as events. This allows your apps to react and respond to new events as soon as they happen, underpinning the shift to a real-time business.

How are event-driven applications different?

In traditional request/response systems, applications directly poll one another to retrieve the latest data and events. This point-to-point design pattern creates tightly coupled software architectures bound by complex interdependencies. The resulting systems are brittle and hard to evolve as new business requirements emerge.



Lack of agility is further hampered by lack of speed. Delays in being able to act on new data are introduced. This is because a consuming application has to poll a producing system to retrieve the latest events (data). Not only does this polling add latency and overhead to each system, it also means new data is consumed and processed in batches, not as soon as it is produced.

Figure 1: *Limitations of request/response systems: hard to evolve, slow to react to new data*

Event-driven applications are different. They invert the request / response model with upstream producer services immediately notifying downstream consuming services as soon as new data (events) are generated. Streams of events are continuously exchanged as messages between systems as they happen – and in the order they occur – not later in batches. The exchange happens through a centralized event streaming platform such as Apache Kafka.

This architectural approach creates an experience that is truly responsive and reactive, while incurring lower system overhead and cost.

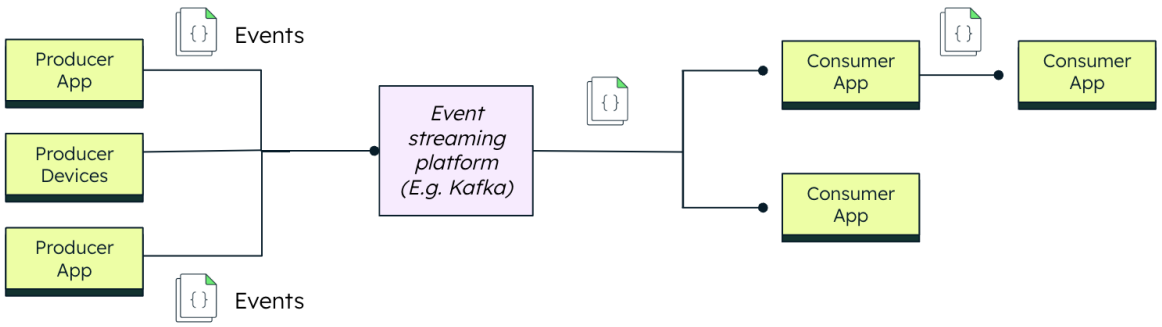


Figure 2: *Data flows across the business are captured and processed as events as they happen*

By design, event-driven applications are loosely coupled, providing your developers a more responsive and agile foundation on which to drive innovation. This is because:

- Each service adapts and evolves independently as needed. They are owned by autonomous teams that are free to work with their preferred technologies and frameworks. Those teams have the flexibility to prioritize their own backlogs based on the needs of the business, not dependencies with other teams.

- Further eliminating dependencies, each service can operate asynchronously from others as needed, and can be deployed and scaled independently.
- Application resilience improves as the failure or maintenance of an individual service does not result in a complete application outage.

Event-driven applications promise a lot. But how your developers work with events across the application stack – in code, in the streams that connect the data flows between your applications, and in storage – will dictate whether you succeed or fail.

The evolution of event-driven technology

Events are the foundation of event-driven applications. An event is an immutable piece of data recording a fact or change in state in an application. For example, a customer placing an order *is an event*. A connected vehicle transmitting its current geolocation *is an event*. A successfully processed payment *is an event*. Some events can be represented in a simple data structure. Many, however, are rich and complex objects that need to be reliably transported, processed, and persisted in order to maintain full fidelity and accuracy.

Applications have always relied on the exchange of events with one another in order to serve business processes and users. With the arrival of microservices and cloud computing, this was typically achieved with events packaged as messages and exchanged via pub/sub (publish/subscribe) queues. In more recent years, the pub/sub queue has largely been displaced by event streaming platforms such as Apache Kafka. Offering advances in message persistence, delivery and ordering guarantees, along with improvements in scalability and resilience, more and more developers have selected these stream messaging platforms.

We can think of the event streaming platform as providing the plumbing to communicate and transport events between different applications and microservices. Events are persisted in a message log, while message brokers along with consumer and producer APIs work in concert to reliably exchange events between independent services.

For events to be useful, the consuming system needs to process and react to them. Event processing can be something simple like counting the number of impressions an online ad receives or recasting a data type in a message. It can also be way more complex; for example detecting fraudulent payments, aggregating sensor events from a production line to predict failures, or extracting and transforming features from clickstream data to serve personalized offers to customers.

Developers have traditionally relied on one of two approaches to event processing:

1. **In-app event processing**. The developer writes all of the computational logic needed to process events as part of their application code, along with mechanisms to manage interim state. As processing becomes more sophisticated, developers face massive increases in complexity. They have to write highly intricate code that is hard to debug, optimize, and maintain.
2. **In-database event processing**. Rather than write app-side code, developers land events directly in the database and use its native query capabilities to process the data – assuming the database supports the query operators that are needed. While simplifying app code, additional latency is incurred as each event has to be ingested, indexed, and persisted before it is available for processing. Databases are also optimized to work with bounded batches of data at-rest in storage, not with boundless streams of data in-motion.

Another factor to consider is the ever increasing volumes of data and events generated by modern applications – aka as *"the firehose"*. Even with backpressure offered by the streaming platform, these volumes can still risk overwhelming the ingestion and processing capabilities of even the most well written apps or fastest databases.

To try and address some of these challenges, several stream messaging platforms offer basic transformation functions. These are typically limited to reshaping or reformatting individual messages. Anything more advanced needs to be written by the user as a custom function, incurring much of the same pain described for in-app event processing.

Event stream processing was born to address the multi-faceted challenges described above. Stream processors do this by continuously processing streams of events consumed directly from streaming platforms such as Kafka. Processing can take the form of filtering, querying, aggregating, transforming, alerting, and routing events.

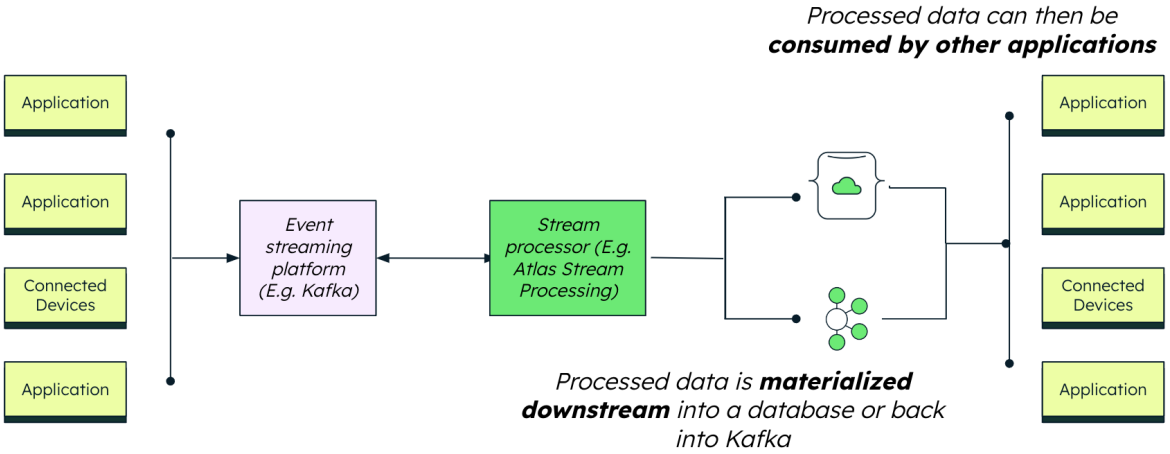


Figure 3. *Event streaming and processing data flow*

Stream processing isn't required by every event-driven application. But increasingly users expect the applications they rely on to be fast, scalable, and responsive, making stream processing an essential part of the stack. But stream processing is still a relatively new technology domain that comes with some growing pains.

Fragmentation + friction = frustration

Today, developers can choose from a number of mature event streaming platforms and databases to handle the transport and persistence of events. However, processing high volume streams of events as they are in-motion is another matter, and one that developers are struggling with.

They are struggling because processing data in-motion is different from working with it in the application and at-rest in the database. Developers have to contend with differences in languages, APIs, drivers, and tools. All of these differences create a fragmented experience which slows them down, adding time, cost, and complexity to building and evolving the application.

Where do all of these differences come from?:

- The native programming APIs provided by today's stream processors are typically Java-based – so you are out of luck if your app is developed in something else or your skillset is rooted in a different language. And then the API you use to persist and process data in the database is entirely different from the stream processor's API.
- To try to reduce API fragmentation, some stream processors offer a SQL interface. But this forces developers to project a rigid, tabular data model against the event's rich and complex schema. Now it bears no resemblance to the event object they work with back in application code, creating yet more cognitive dissonance.
- A rigid schema also makes it much more difficult to handle sparse data and schema changes demanded by new application requirements. Schema changes need to be coordinated across the app, ORM layer, stream processor, and database – while trying not to disrupt high velocity streams of data flowing into your systems. All of this increases the time, risk, and cost of rolling new features to customers.
- Finally the stream processor is itself a separate technology that needs to be bolted into the application stack. Now the developer has to contend with another set of drivers and tools, along with a completely different set of operational and security concerns. Maintaining application state across the stream processor and database becomes incredibly complex, especially in the face of failures.

All of these issues add friction and frustration to the development of event-driven apps. Each on their own consumes valuable cycles that would be better spent on innovating for the business. Taken collectively, they risk inhibiting the wholesale adoption of stream processing and event-driven applications.

MongoDB Atlas gives you a better way. That is because events map naturally to flexible JSON-like document data structures and are best processed by an expressive and composable query API. Both of these constructs are core to how developers work with MongoDB, making it one of the most widely used data technologies on the planet today.

A Developer's Path to Productivity

The MongoDB Atlas database has long been used as a database foundation for event-driven apps. With the introduction of Atlas Stream Processing, developers have – for the first time – a completely unified experience working with data across the application stack, spanning data in-motion and data at-rest.

MongoDB's flexible document data model and expressive, developer-native query API gives developers a consistent way of working with data in the stream and in the database. Breaking down these barriers transforms the experience for developers building new classes of real time, event-driven applications.

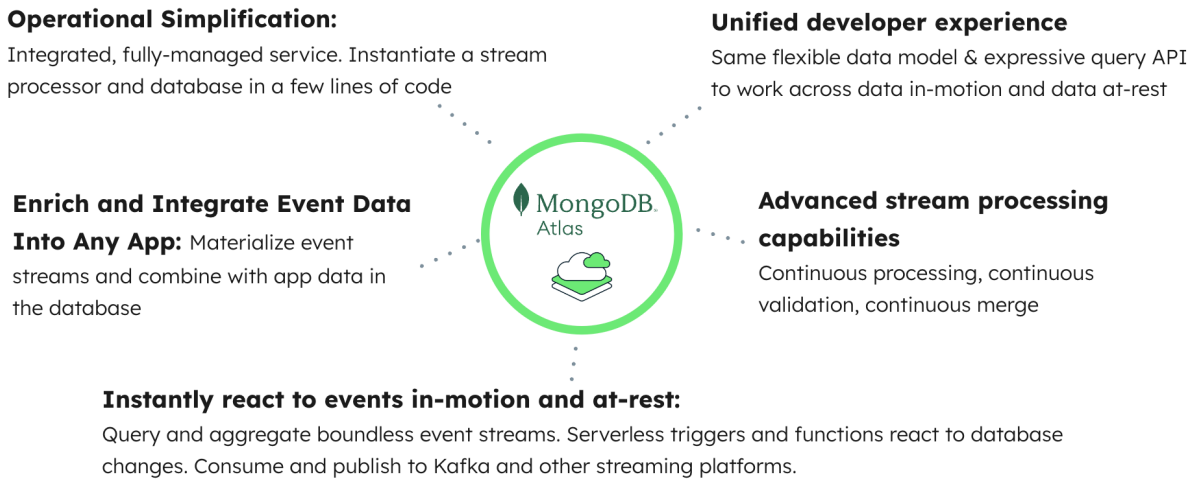


Figure 4: *MongoDB Atlas transforms how developers build event-driven applications*

By building on Atlas, developers have everything they need to create apps that instantly react to live events. Those apps can deliver deep insights against a firehose of streaming data, allowing you to immediately respond to opportunities and detect threats. They can combine events with stored application data, searching and enriching it to drive operational processes across the business. And then those events can be archived out of live systems as they age. All on top of a foundation that is

fully-managed with built-in data security and redundancy defaults, along with deep observability into operations.

Atlas Stream Processing is part of MongoDB Atlas, the multi-cloud developer data platform. Atlas combines transactional processing, stream processing, application-driven intelligence, relevance-based search, and mobile and device edge computing with cloud sync. All delivered in an elegant and integrated data architecture that powers almost any class of application.

Enabling Technology for Event-Driven Apps

Founded on flexible documents and an expressive query API, MongoDB Atlas offers developers the key capabilities they need to build powerful and innovative event-driven apps.

Document Data Model

Documents are the best way for developers to build event-driven apps. This is because documents are:

1. **Natural.** Unlike traditional tabular data structures, documents map to event objects in code and make it straightforward to represent complex event structures in-motion in the stream and persisted to the database. This consistency across the stack means documents are much more natural for developers to work with and eliminate any opaque mapping layers that are otherwise required to wrangle data structures between different layers of the stack.
2. **Flexible and validated**. Developers can easily handle sparse data inherent in stream processing and modify document structures at any time as app requirements evolve. This eliminates the delays and dependencies that come from having to update rigid data models across ORM layers, schema registries, and databases. Ensuring data integrity, MongoDB's schema validation ensures documents are properly formed before being processed by Atlas Stream Processing or persisted into the Atlas database.
3. **Versatile and extensible**. Developers can model data and events in any structure to support almost any application requirement - hierarchical objects typical of JSON records, key-value pairs, geospatial coordinates, time-series measurements, embedded relationships, the nodes and edges of a graph, and more.
4. **Blazing fast**. Documents bring data together that is accessed together. This reduces the need for complex joins, simplifies query logic and planning, and ensures low latency reads and writes – essential for real-time apps.

MongoDB Query API, Language Drivers, and Tools

In the same way documents provide a consistent way of representing data in the stream and database, so the MongoDB Query API and tools provide a consistent way of working with that data in-motion and at-rest.

The MongoDB Query API and its aggregation framework is implemented in the methods and functions of native programming languages. For developers, this makes the query API syntax feel like an extension of their chosen language, enabling them to work productively with data as code. Whatever the preferred languages and frameworks, there are three further design constructs that simplify event-driven app creation for developers:

1. The MongoDB drivers are available with asynchronous programming support for all of the leading development languages.
2. Implemented as a composable pipeline of processing stages, the aggregation pipeline will immediately feel familiar to any developers with prior stream processing experience.
3. Tools such as the MongoDB Shell and MongoDB Compass – the GUI for MongoDB - make it straightforward to author and run stream processing pipelines.

The aggregation framework is one of MongoDB's most powerful capabilities - we discuss more of what you can do with it in the Atlas Stream Processing section below. In addition to product documentation, the Practical MongoDB Aggregations ebook is a great resource to help developers get started.

MongoDB Atlas Stream Processing

Atlas Stream Processing enables developers to work with high-velocity streams of complex event data using the same data model and query interface that's used for their database.

Atlas Stream Processing is part of the MongoDB developer data platform. With this integration, developers can instantiate a stream processor, a database, and API access layer in just a couple of API calls and lines of code. Along with the other data services in MongoDB Atlas, developers will be able to use Infrastructure as Code tools to provision, manage, and control Atlas Streams as part of their continuous delivery workflows.

Atlas Stream Processing is built around three key capabilities that accelerate the delivery of new event-driven apps:

1. Continuous Processing: Developers use MongoDB's aggregation framework to process rich and complex streams of data from event streaming platforms such as

Apache Kafka. This unlocks powerful new ways to continuously query, analyze, and react to streaming data without any of the delays inherent in batch processing.

With the aggregation framework you can filter and group data, aggregating high velocity event streams into actionable insights over stateful time windows and powering richer, real-time application experiences.

2. Continuous Validation: Atlas Stream Processing offers developers robust and native mechanisms to handle incorrect data issues that can otherwise cause havoc in applications. Potential issues include passing inaccurate results to the app, data loss, and application downtime. Atlas Stream Processing defends against these issues to ensure streaming data can be reliably processed and shared between event-driven applications. Atlas Stream Processing provides:

- Schema validation to check that events are properly formed before processing – for example rejecting events with missing fields or containing invalid value ranges.
- Detects message corruption or late arriving data that has missed a processing window.

Atlas Stream Processing pipelines can be configured with an integrated Dead Letter Queue (DLQ) into which incorrect data is routed. This avoids developers having to build and maintain their own custom mechanisms. Issues can be quickly debugged while the risk of missing or corrupt data bringing down the entire application is minimized.

3. Continuous Merge: Processed data is continuously pushed into a materialized view maintained in an Atlas database collection. We can think of this as a push query. Applications can retrieve results (via pull queries) from the view using either the MongoDB Query API or Atlas SQL interface.

Continuously merging updates to collections is a really efficient way of maintaining fresh analytical views of data supporting automated and human decision making and action. We illustrate the power of Continuous Merge and materialized views in the event-driven apps examples later in the paper.

In addition to materialized views, developers also have the flexibility to publish processed events back into a topic in the event streaming platform for consumption by downstream systems.

Atlas Stream Processing will shortly be available for preview. You can apply to be part of the preview program here.

MongoDB Atlas Database

MongoDB has grown to become the world's most popular modern database. The document data model and query API are built on a transactional storage engine to support almost any class of operational workload. These workloads can be augmented with powerful database-native application-driven analytics to drive real-time actions and insights from live data.

The MongoDB Atlas database runs on a distributed and elastic architecture providing horizontal scale-out and built in redundancy. With Atlas, users can support the largest workloads with a 99.995% uptime SLA.

Developers can run MongoDB anywhere, with Atlas available as a managed database service across 100+ regions and multi-cloud deployments. It provides always-on security with operational best practices and performance optimizations all baked in.

MongoDB Connector for Apache Kafka

Kafka Connect is a component of Apache Kafka that makes it easy to integrate data sources such as MongoDB within the Kafka ecosystem. The MongoDB Kafka Connector is an open sourced connector used by thousands of organizations today. It runs in the Kafka Connect framework, enabling Atlas to be used both as **a source and a sink**. This means events from Kafka topics can easily be consumed in Atlas and data from Atlas can be written out to a Kafka topic for consumption by downstream systems. The connector is developed and supported by MongoDB and verified by Confluent.

MongoDB Change Streams, Triggers, and Functions

Change Streams are a change data capture mechanism built into the MongoDB database. Through change streams, applications can subscribe and react to real time data changes in the database as they happen.

While applications can directly subscribe to change streams, Atlas Triggers provides a serverless way of consuming change stream events. With Triggers, developers don't have to stand up their own application server to run the change data capture process.

Each Atlas Trigger is linked to an Atlas Function that invokes server-side logic in reaction to relevant events. Like Atlas Triggers, Functions are serverless, so you only pay for them when they are running. Atlas Functions make it easy to implement application logic, securely integrate with cloud services and microservices, and build APIs.

Driving innovation with event-driven apps

With the enabling technologies described above, developers can build new classes of event-driven applications. In this section of the paper we focus on 4 key use cases enabled by MongoDB Atlas:

1. Event-driven microservices.
2. Event-driven streaming analytics.
3. Event-driven operational data layer.
4. Event-driven ETL.

1. Event-driven microservices

Integrating independent microservices to power a business application is one of the most common use cases for an event-driven architecture.

Figure 5 below shows an ecommerce platform. A user placing an order generates an event which is published to a Kafka topic. This makes it available for consumption by the other microservices needed to process and fulfill the order – for example, payment processing, inventory management, pick list generation, shipment, etc.

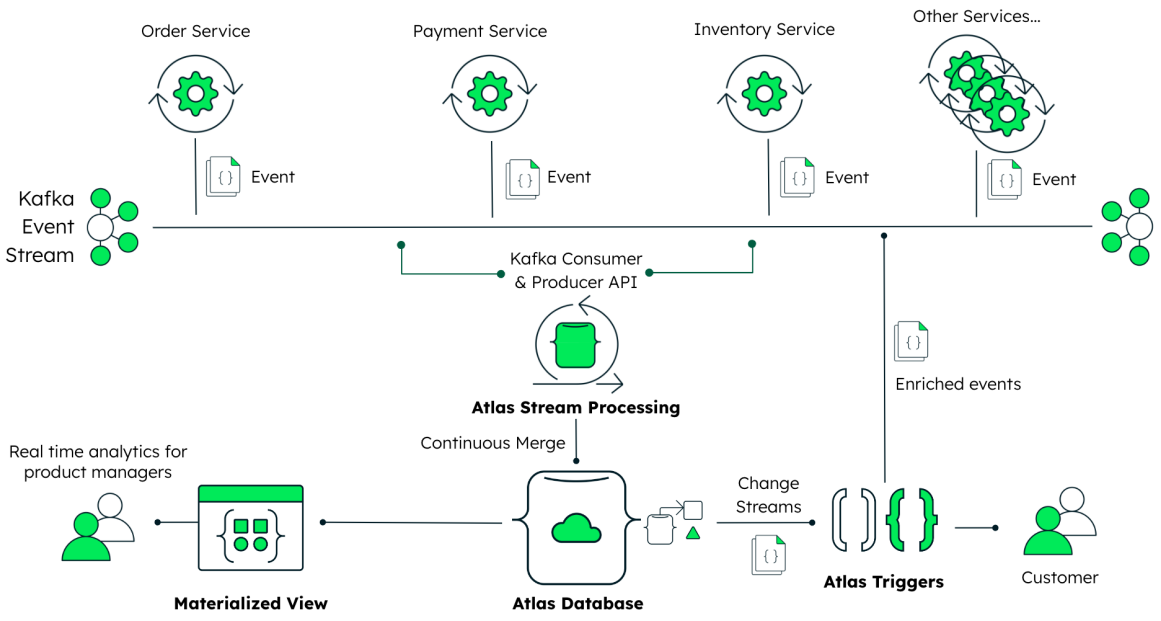


Figure 5: *Processing and fulfilling orders through a suite of ecommerce microservices while powering real time business visibility*

Atlas Stream Processing continuously processes streams of events while they are in-motion between microservices. In our example, the payment processing service checks for potential fraud by using Atlas Stream Processing to query each order event as soon as it is generated, providing a low latency check-out experience for the customer.

Approved orders are inserted to the orders collection in the local Atlas database. On insert of each order, a MongoDB change stream is fired that automatically triggers an email acknowledgement to the customer. Approved orders are also enriched with customer data retrieved from the Atlas database, and then published back to a Kafka topic for consumption by the downstream order fulfillment services.

Atlas Stream Processing aggregates orders as they progress through the ecommerce system, merging and materializing results into an Atlas database collection. Business managers access these continuously updated materialized views to track sales and inventory by product line and region. With this real-time visibility, the business can dynamically adjust pricing, launch flash sales, and optimize their supply chains all in response to real-time market demands.

Note that for readability, only one Atlas database is shown Figure 5 above. In a microservices architecture it is common to segregate data per microservice or per domain. MongoDB provides engineering teams with the flexibility to select the isolation model most suited for the needs of the application.

2. Event-driven streaming analytics

As noted in the introduction section, applications must increasingly be able to analyze and react in real time to live events as they happen. We can no longer afford the delay in first having to ETL events into a data warehouse or data lake before then being able analyze them.

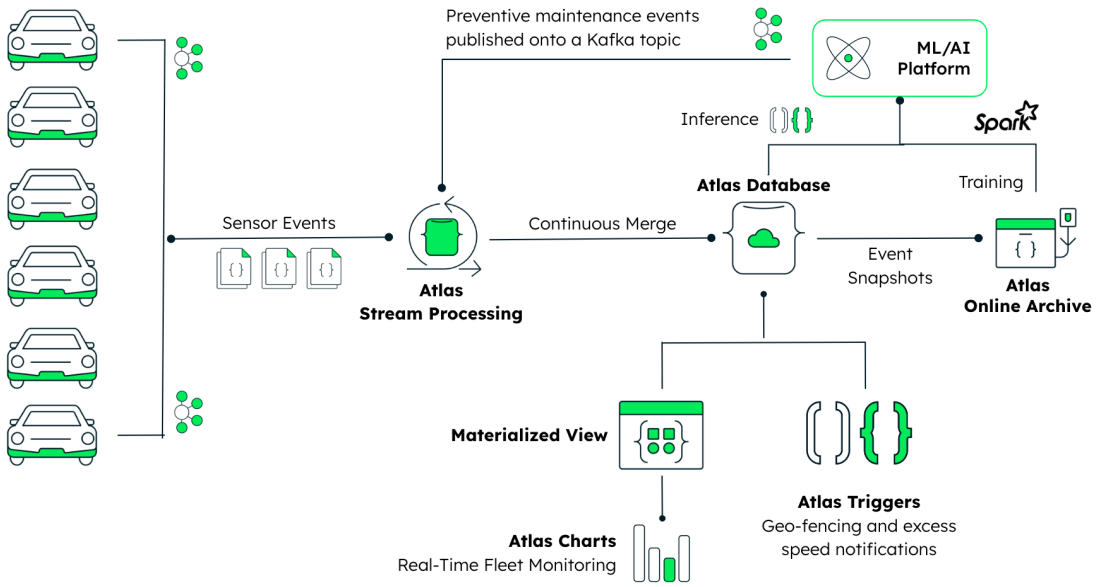


Figure 6: *Application-driven streaming analytics enabled by Atlas*

Figure 6 demonstrates a fleet management application powered by MongoDB Atlas. Streams of sensor data generated by each vehicle are published into Kafka topics. Atlas Stream Processing computes and transforms this raw sensor data, materializing

it in Atlas database collections for real time fleet monitoring and dashboarding. These views enable fleet managers to track a variety of signals including vehicle locations and route mapping, fuel consumption, driver behavior, vehicle utilization, and more. Anomalous events such as geo-fencing and excess speed alarms can be instantly alerted and actioned.

Aged events are automatically tiered into Atlas Online Archive, with the MongoDB Spark Connector used to serve them into machine learning model training. Trained models are exposed to the application via an API endpoint which is invoked by an Atlas Trigger when potential service issues are detected from processed vehicle telemetry.

3. Event-driven Operational Data Layer (ODL)

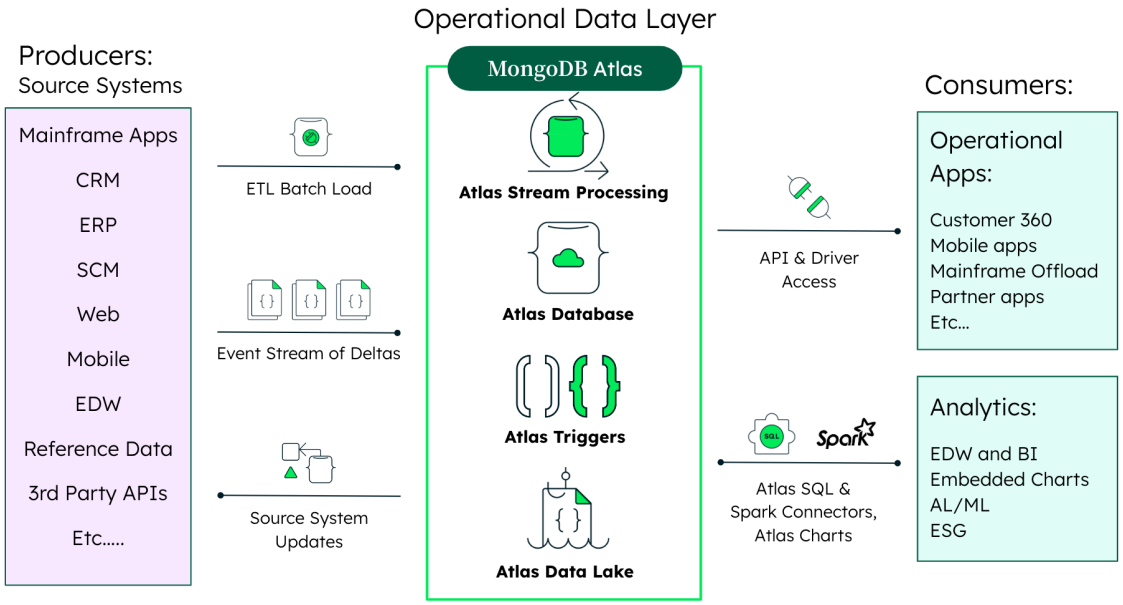


Figure 7: *Liberating siloed enterprise data, making it accessible in real time to modern applications, microservices, and analytics*

ODL is a common pattern used by many organizations to make enterprise data available to new applications and services. What is key to this pattern is that the ODL exposes the data **without** disrupting the existing backend systems that generate it. This approach offers a phased path to digital transformation and legacy modernization.

With its flexible document data model and event-driven capabilities that instantly react to new data in-motion or at-rest, MongoDB Atlas is often chosen as the real-time data layer powering the ODL. Common examples include offloading data from mainframes into an ODL to power new digital services. Consolidating data from multiple operational systems into a single, 360-degree view of the customer is also

common. The 360-degree single view is used to support customer service or power predictive cross-sell and upsell ML models.

To enable the ODL architecture, an event producer is typically added to the source system. The event producer often reuses log files that fire change data capture events to the MongoDB Atlas-based ODL. When first connected to the backend systems, the ODL is initially hydrated via an ETL batch data load. Keeping the systems in sync, change events are then streamed as deltas to the ODL whenever data is written to the source system.

When new data hits the ODL, Atlas Stream Processing and the Atlas database process the data – often consolidating it into a single schema. Atlas will then generate notifications to downstream services that need to consume the data.

4. Event-driven Extract Transform Load (ETL)

Event-driven ETL is a derivative of the ODL and event-driven analytics patterns discussed above.

Live operational data is moved from the application stack into a backend centralized analytics data warehouse or data lakehouse. Processed events are either published directly by Atlas Stream Processing into a Kafka topic for consumption by the analytics system, or triggered from the Atlas database by a change stream. Once the event data from MongoDB has landed in the analytics system, it is combined with other operational data sources for BI reporting and for machine learning.

The outputs of these analytics processes are often loaded back into MongoDB Atlas. From here, intelligence and insight can be served at scale within the flow of the application to its users.

All of Atlas's cloud provider partners have tools and reference architectures helping customers implement event-driven ETL today. To learn more about the current reference architectures, refer to the links below:

- MongoDB Atlas and Amazon Redshift
- MongoDB Atlas and Azure Synapse Analytics
- MongoDB Atlas and Google BigQuery using Dataflow

MongoDB in Event-Driven Apps Today

MongoDB is already widely used for event-driven applications. A few examples follow.

Customer	Use Case	Results
7-Eleven	Digital wallet and mobile enabled inventory and ecommerce	Enables new digital services for customers, improves accuracy of inventory data and raises employee productivity
Keller Williams	Property publishing pipelines, search, and analytics	Sales volumes increased 40% compared to 28% industry average
Rent the Runway	Warehouse automation: garments are x-rayed triggering cleaning and repair processing	Reduced warehouse processing time by 67%
Bosch	IoT: sensor data collection and event-driven analytics	Bosch IoT Suite powers the gamut of industrial, smart city, and smart home applications
EnBW	Management of both charging points and customer access with real time data and alerts	Expanded to become one of Europe's largest EV charging point networks
Humana	Cloud-native ODL for FHIR implementation	Enables data sharing with providers and patients, along delivery of new wellness services
Toyota Financial Services	Mainframe offload via ODL: fraud detection, customer onboarding	*"MongoDB helps us make better decisions and build better products."*
Midland Credit Management	Event-driven ODL to build a single view of the customer	50x higher scalability, 120x lower costs, zero data errors
Slice	Event-driven ML feature store for consumer credit underwriting	Reduced credit application process from 48-hours to 30-seconds

Getting started with event-driven apps

The best way for your developers to get started is to sign up for an account on MongoDB Atlas. From there, they can create a free database cluster with change streams and Atlas Triggers, load their own data or our sample data sets, and explore what's possible within the platform. Atlas Stream Processing is available to select developers today via a Private Preview program.

The MongoDB Developer Center hosts an array of resources including tutorials, sample code, videos, and documentation organized by programming language and product. We also offer self-paced training via the MongoDB University, along with instructor-led training and consulting services delivered by MongoDB Professional Services.

Collectively, these resources help you get started on your event-driven, real-time journey!

Safe Harbor

The development, release, and timing of any features or functionality described for our products remains at our sole discretion. This information is merely intended to outline our general product direction and it should not be relied on in making a purchasing decision nor is this a commitment, promise or legal obligation to deliver any material, code, or functionality.

US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com.