# ANALYTICS ARCHITECTURE FOR THE MODERN SOFTWARE DEVELOPMENT ORGANIZATION

source allies

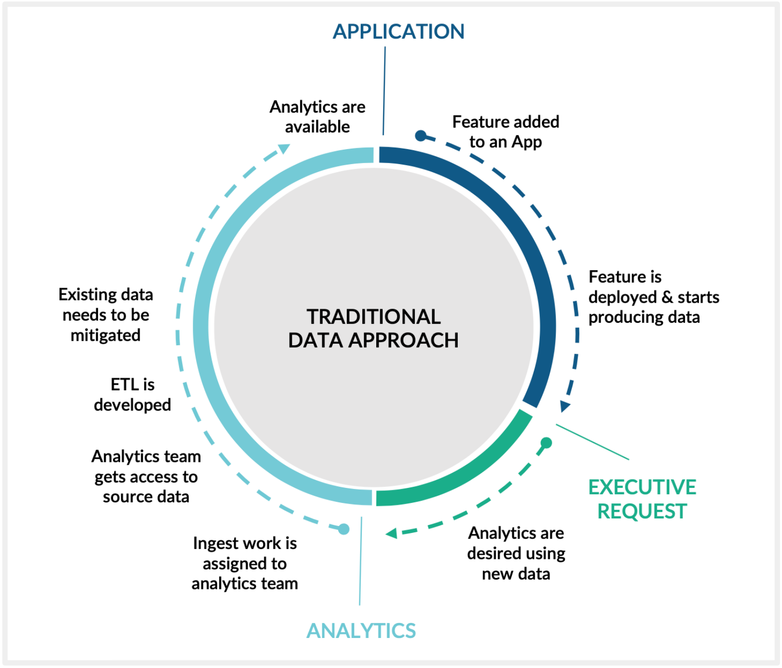
# Analytics Architecture for the Modern Software Development Organization

## EXECUTIVE SUMMARY

In many of today's organizations, a disconnect exists between the teams involved in the production of data and the teams involved in the interpretation of data. Data analysis is often secondary, and the traditional model is costly and time-consuming to maintain. Businesses need a newer, more agile approach to keep up with innovation and the growing demand for real-time data analytics. This technical white paper proposes an alternative solution that reduces expenses and improves flexibility, while still leveraging the technical experts within an organization. The idea is to create a paradigm shift that's pro-education, pro-decentralization, pro-self-service, where application development teams become responsible for making their data analyzable from the beginning. That data should then be broadcast in such a way where it can be consumed by downstream systems.

## THE TRADITIONAL APPROACH

In the typical product life cycle, several independent applications are built over time for any given organization. These applications are responsible for housing mission-critical data. Once these applications are deployed, a project is started to unite all of this data to gain business insights and drive decision-making. Sometimes an expensive data-warehouse and ETL package; or data lake solution is purchased and deployed. This can create problems down the line for most companies because the attempt to integrate with these systems happens *after* the large capital expenditures. In addition, integration generally consists of pulling all data every day or using some large flat-file transport. But what about data latency? Does new data need to be available for analysis weekly, daily, or even hourly? Can a tremendous batch job be sustained when running this frequently? Never mind that performant applications today may not be performant a month from now.


APPLICATION
Analytics are available
Feature added to an App
Feature is deployed & starts producing data
TRADITIONAL DATA APPROACH
Existing data needs to be mitigated
ETL is developed
Analytics team gets access to source data
EXECUTIVE REQUEST
Analytics are desired using new data
Ingest work is assigned to analytics team
ANALYTICS

These traditional systems are expensive to build and difficult to maintain. There are many upfront costs, and it could take an entire team of dataops, data analysts, and systems engineers to find data, transform it, and put it into a data warehouse or data lake. The tools used to accomplish this are typically not amenable to test-driven development or deployments without human interaction. These systems do not usually use industry-standard languages and frameworks or Continuous Integration (CI) or Continuous Delivery (CD) best practices.

The maintenance and upfront development costs extend beyond the initial purchase of these tools as well. For example, part of your license may include a consultant from the vendor who comes in to modify the source code of your installation. The problem is that customizing your installation makes it difficult to upgrade when a new version is available from the vendor.

## CONSTANTLY PLAYING CATCH-UP

When a source system starts producing an additional piece of data, multiple teams need to get involved, using multiple technologies, to get a single piece of data in the warehouse. This is a catch-22 in the sense that new analytics requirements slow down the release of application features, and changes after the release require the back-population of data that was recorded before the change to support analytics. Without a complete audit trail, insights that may have been available during the initial release may be lost. On top of this, mergers and acquisitions don’t stop due to a data warehouse project, so those new data sources have to be integrated as well.

As we move toward being more data-driven, our data sources change while we're trying to develop a centralized data insights solution. Organizational and product innovation does not stop while we build centralized data insight engines. Not only does the scope and scale of data continuously increase with time, but the needs of new performance indicators change at the pace of product and service innovations. Batched, centralized ETL data warehouse and data lake solutions, no matter the level of investment, cannot keep up with the pace of today's business innovations. This line of thinking is also demonstrated in a recent ThoughtWorks article by Zhamak Dehghani:

> *"The assumption that we need to ingest and store the data in one place to get value from a diverse set of sources is going to constrain our ability to respond to proliferation of data sources. We have created an architecture and organization structure that does not scale and does not deliver the promised value of creating a data-driven organization. In order to decentralize the monolithic data platform, we need to reverse how we think about data, it's locality and ownership. Instead of flowing the data from domains into a centrally owned data lake or platform, domains need to host and serve their domain datasets in an easily consumable way."*

## THE SOLUTION

Our solution focuses on flexibility and agility. We assume that tomorrow you'll be more knowledgeable than you are today, and we believe it's important to focus on avoiding one-size-fits-all schemas. Rather than analytics and ETL being the sole responsibility of a specific set of tools distinct and different from the systems that are producing the data, these become first-class features of producing systems. Certain major companies like Amazon have already started implementing this approach. In an email memo that Jeff Bezos sent to his employees, he stressed:

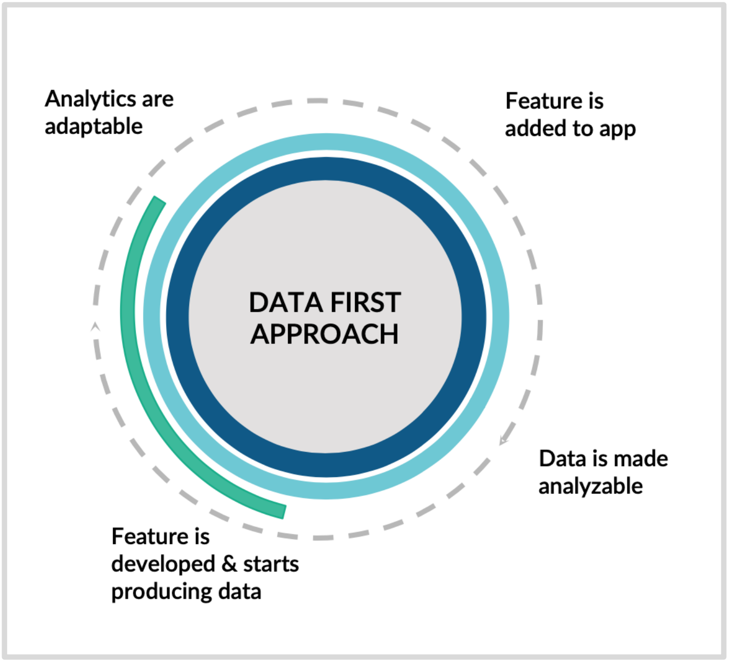
> *"1) All teams will henceforth expose their data and functionality through service interfaces.*
>
> *2) Teams must communicate with each other through these interfaces.*

*3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*

*4) It doesn't matter what technology is used. HTTP, Corba, Pubsub, custom protocols — doesn't matter.*

*5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions."*

As an application that is producing new data for an organization (e.g., user facing data entry), it is that application's responsibility to make the data that it produces consumable to others without having to retrofit the application. This goes for systems that exist today, as well as future systems that may be built. At a high level, this is achieved by broadcasting an event to interested downstream systems when activities occur, and ensuring that all data is available using that event even if there is no consumer of that data today. This is a scalable solution because as systems become more connected, the complexity of the source system does not increase and consuming applications can have full control over what they do with the data they are getting.



## THE ROLE OF PRODUCERS AND CONSUMERS

There are two types of consumers: one is business-driven, the other one is state-driven. Business-driven consumers care about the "why" and want all the details (e.g., when, where, and why a customer signed up). State-driven consumers just need to know when a customer changes, but don't necessarily care why. However, both care about the “what.”

What producers typically do:

- Process transactions
- Decide what the event should contain (e.g., pointers to source data, or metadata about the action or data recorded)
- Decide whether the event should be a state-change event or a business-level event

What consumers typically do:

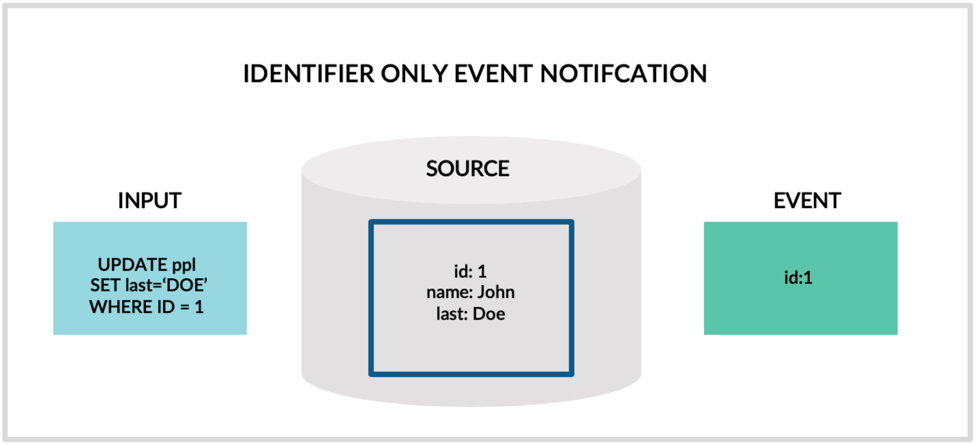
- Window and filter anything that requires action
- Track key performance indicators (e.g., an aggregate of the total sales per month)
- Use the correct data structures and algorithms to fit the distributed nature of event-driven systems and their unique memory and consistency requirements. An example of this can be found in Andrii Gakhov’s most recent book, *Probabilistic Data Structures and Algorithms for Big Data Applications*. In it, he writes:

> *"Probabilistic data structures is a common name for data structures based mostly on different hashing techniques. Unlike regular (or deterministic) data structures, they always provide approximated answers but with reliable ways to estimate possible errors. Fortunately, the potential losses and errors are fully compensated for by extremely low memory requirements, constant query time, and scaling, the factors that become essential in Big Data applications.”*

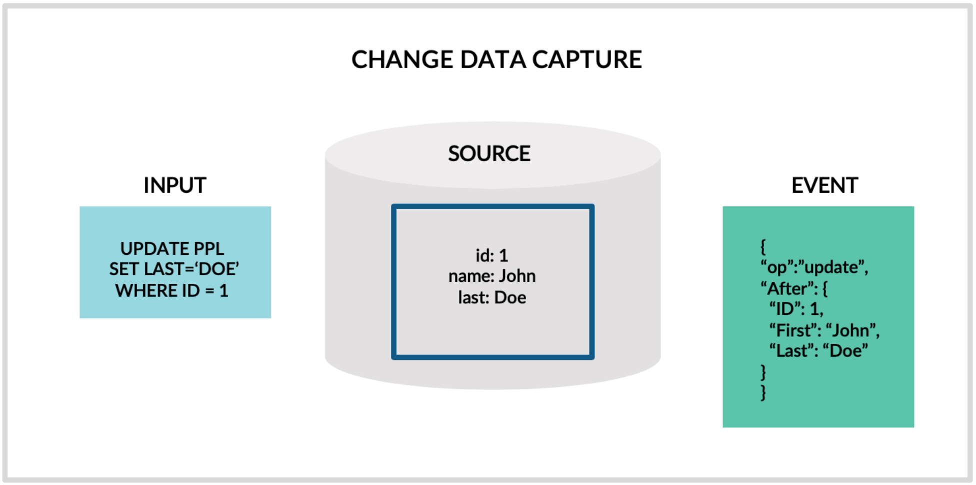
## TYPES OF BROADCASTING

When producers are broadcasting information, they should be doing it in such a way where the information can be consumed by any downstream system. The question we need to ask ourselves is: What are we producing and consuming? For each logical topic, determine when events should be broadcast and what they should contain. Some high-level categories to consider are:

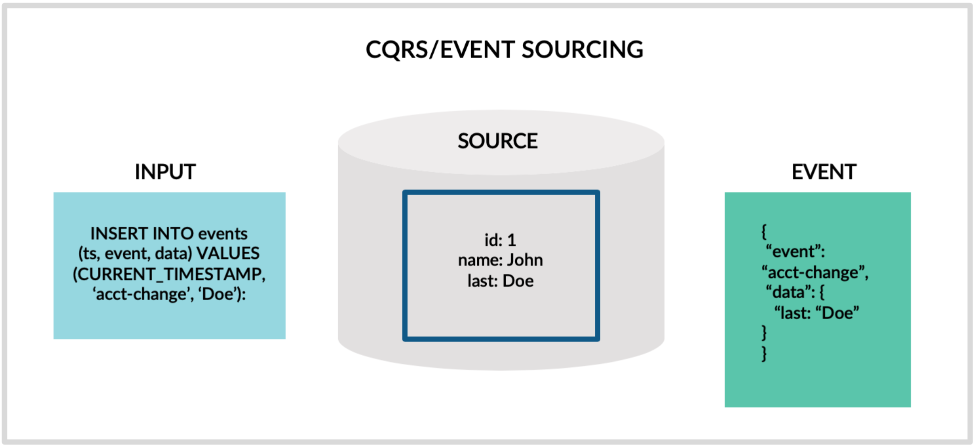
- Broadcast a change notification with the primary key/ID of the record from the source system: Don’t put the original nor the changed data in the event. This solves lots of problems if you can support the load and you're okay with latency. However, this can cause loss of resolution if the data changes fifteen times before the interested applications decide to pull the most recent version of the data after receiving the event notification.


IDENTIFIER ONLY EVENT NOTIFCATION
INPUT
UPDATE ppl SET last='DOE' WHERE ID = 1
SOURCE
id: 1
name: John
last: Doe
EVENT
id:1

- Change Data Capture (CDC): With this strategy, you can capture changes without changing the application and technologies. You don't even have to impact the performance of the application because all you're doing is tailing the write-ahead log (with, as an example, Debezium: https://debezium.io/). The upside to this approach is you don't need to change code for existing systems connected to databases. The downside is that you only see events that involve a database change (as opposed to user gets, actions that don't get captured). Also, you'll see that data changed, but you won't necessarily see why it changed.



- CQRS/Event Sourcing: Rather than building a system where business processes are saved in a database via an update statement, instead, consider using an insert only data structure. This way, your entire system is built on events that occur versus an after effect of state. Current state is always derivable from the event log. Another upside to using this approach is the “why” is captured and the events are all idempotent. A drawback is it requires change in mindset over the traditional development mindset. It’s such a fundamental change that it’s a rewrite for existing applications, not something you just add in or layer on. Another con is this requires high coordination with the business. Events should be based on the business process and transitions between nodes/steps in a flowchart. This takes a lot of communication. However, it provides a lot of value in compliance, regulatory requirements, etc., because all of the regulatory stuff becomes free. (If you’d like to know more about this approach, check out Source Allies' Event Sourcing blog here: https://www.sourceallies.com/2019/11/event-sourcing/.)


CQRS/EVENT SOURCING
INPUT
INSERT INTO events (ts, event, data) VALUES (CURRENT_TIMESTAMP, ‘acct-change’, ‘Doe’):
SOURCE
id: 1
name: John
last: Doe
EVENT
{ “event”: “acct-change”, “data”: { “last: “Doe” } }

Keep in mind that when considering each of these strategies, it’s almost always necessary to know when an event occurred. This is why we strongly recommend including the event time. This way, the downstream systems can decide what to do with necessary information. It’s also important to ensure that the transactions are consistent. No matter which broadcast strategy you use, an application should not broadcast an event and roll back a transaction, or commit a change to the data, without broadcasting an event.

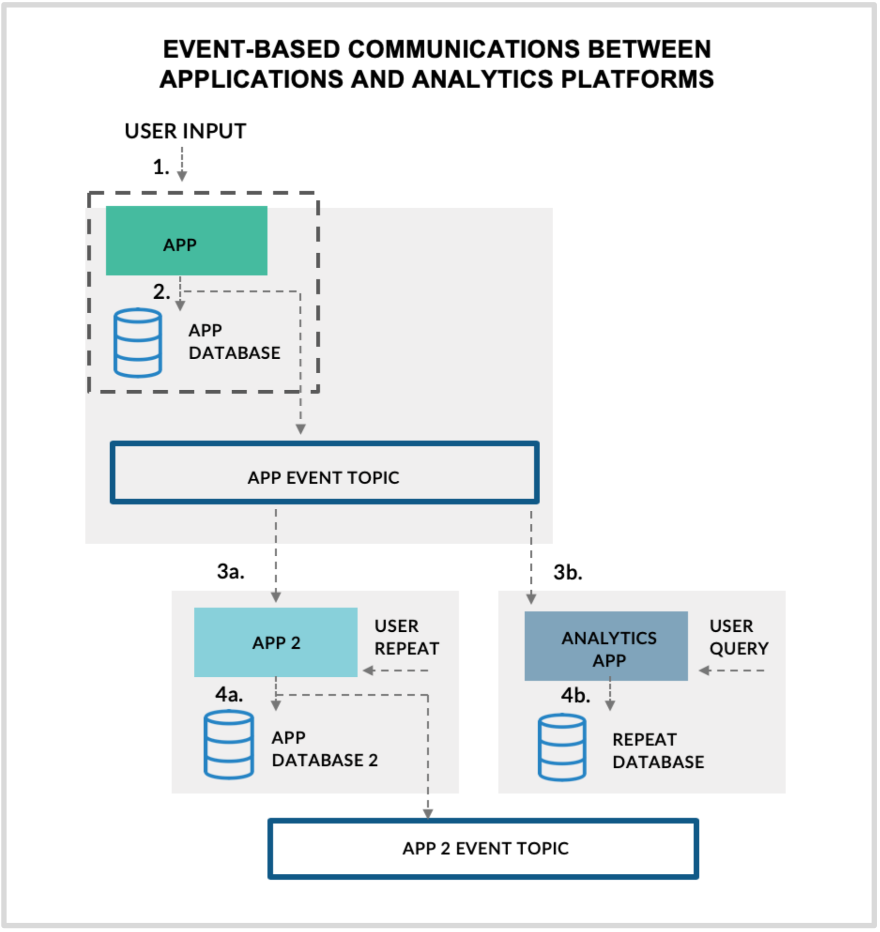
From the consumer side of things, the paradigm shifts from “getting all the data” on a schedule or having all the data locally to instead reacting to business activity via the events that producing systems are broadcasting. For example, rather than holding a copy of every user account or running a query against the whole accounting system every night to look for users that have signed up within the last day, we could implement something closer to how the business describes the requirements. “When a user signs up, we want to track what countries we are operating in.” We can listen to the user events from the accounting system and when a sign up event occurs, we can increment a counter or store a record as needed. While this may seem more complicated than batch jobs at first glance, it is actually much simpler. There is no need to manage a scheduler, and we don’t have to worry about a bad record failing the job. We don’t need to have conversations around how often to run it, either. We don’t have performance concerns when querying an ever-growing source data set. We don’t have to worry about missing data because we didn’t include it in our query. We are not highly coupled to the source system’s data storage format. We can scale horizontally in reaction to load.

When consuming events from an event-driven system like proposed above, here are some characteristics to be aware of:

- You may get events out of order (especially between producers)
- You may get an event more than once
- Must beware of inserts/updates on the producer (i.e., data changing between broadcast and consumption)

## TIGHTENING FEEDBACK CYCLES

Given that every system can listen to anything it wants to, every system can serve analytics within its domain. If you broadcast new events in reaction to upstream events, not only is that useful to others, but each individual application can use and listen to their own event stream. This can also reduce the operational load on shared infrastructure. With this approach it is not necessary to rely on a data warehouse to add support or aggregate data in order to gain business insights; applications can self-serve that information. Furthermore, companies don’t need to perform nightly roll ups of transactions internally. Instead, they should consider consuming their own events to do summary and roll up operations, as opposed to performing monster queries via periodic batch processes. Simply consider the type of statistic that your analysis is being reported on and then choose your aggregation strategies and create your consumers accordingly.



The real benefit to this proposed solution comes from scaling out the applications interested in the source data. Instead of looking at a report as a one-off special snowflake, we can classify it as another application. Applications that produce data, broadcast their produced data, and applications that need data, consume it. By decentralizing analytics functionality (aggregations, roll-ups, machine learning, etc.) and skill sets, any team can self-serve its analytics needs. This reduces vendor lock-in because it allows teams to use a variety of technologies at their disposal. It also reduces the dependency on shared analytics infrastructure and teams in order to deliver.

## CONCLUSION

In today's climate, building our applications to broadcast information is a more efficient way to move data between applications. It allows interested parties (e.g., other apps or analytics solutions) to listen to that broadcast and implement their own features in reaction to those changes. The idea is to make every team responsible for building competency in analytics technologies from the very beginning. It's still necessary to have people within our organizations who specialize in data analytics. However, their main job isn't to implement all ingestion and transform the data (pleasant news for analytics professionals who typically spend a majority of their time wrangling data, fixing pipelines, etc); their job is to distribute analytics knowledge and facilitate those capabilities within project teams, similar to how DevOps specialists provide tooling and practices to their teams in order to be responsible for their own applications. This is a transformative approach that allows analytics to scale with your business, saving your company time and money, and allowing you to make sound business decisions based on accurate data.

## ABOUT SOURCE ALLIES

Source Allies is a technology services company based in Des Moines, Iowa. Our multi-disciplinary teams are comprised of IT professionals whose areas of technical and process specialization include software development, cloud infrastructure, user-experience, and information security. We have extensive experience building distributed, test-driven, secure analytics solutions, whether it's legacy or greenfield, on-prem, or with any of the three major cloud providers. Our company takes an iterative, cross-functional, team-focused approach to all development to ensure that we're building the right thing at the right time to solve the right problem. We work closely with our partners to solve mission-critical technical problems and deliver high-quality software solutions.

## REFERENCES

Dehghani, Zhamak. (2019.) "How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh." martinfowler.com. Accessed February 26, 2020. https://martinfowler.com/articles/data-monolith-to-mesh.html.

Biehl, Matthias. (2019.) "The API Mandate — Install API Thinking at Your Company." API-University. Accessed March 12, 2020. https://medium.com/api-university/the-api-mandate-install-api-thinking-at-your-company-4335433b7d0b.

Gakhov, Andrii. (2019.) *Probabilistic Data Structures and Algorithms for Big Data Applications*. Published by Books on Demand. vii, 3–17, 22.