APRIL 2024

Standalone Databases vs. the Modern Data Platform

How the hidden costs and complexity of bolt-on databases are a drag on innovation.

MongoDB.

Table of Contents

Vectors: The latest bolt-on tech

There's a modern gold rush happening and the hot commodity is artificial intelligence. AI is transforming every industry and market segment, from technology and healthcare to customer service and cybersecurity. Anywhere a competitive edge can be gained, someone is using AI to do it. But like the gold rush of the early 20th century, the modern one requires tools and know-how to mine for value that may be many layers deep. Instead of the maps, picks, and pans of old-time gold miners, modern AI practitioners need technology tools to strike it rich.

When it comes to AI there are three layers that must be in place to extract value from the available technology. The first layer is the underlying compute resources (GPUs) and foundation models that are trained on large corpuses of data, usually referred to as large language models (LLMs). The second layer is the tooling used to fine-tune models and build applications that leverage the trained models. The third layer is the AI applications themselves. In this white paper, we're going to focus on the second layer, because that's where modern-day gold miners can tap into the rich vein that will lead them to the riches they seek.

Within that layer lies a collection of tools and techniques for improving the information that AI systems like generative AI use to respond to prompts. When AI relies solely on the corpus of data LLMs are trained on, it will often produce results that are either inaccurate or lack information within a specific business context. This is because LLMs are trained on data that has since become outdated, incomplete, or lacks proprietary knowledge about a specific use case or domain.

One of the techniques used to inform LLMs of this critical contextual data is retrieval augmented generation (RAG). RAG works by combining a pre-trained LLM with a retrieval system of readily accessible information. RAG models enable LLMs to generate more accurate answers with up-to-date and relevant context for the task at hand. The information is stored as numerical representations in high-dimensional space called vector embeddings (Figure 1). These embeddings are capable of capturing the semantic or underlying meaning of specific data sets.

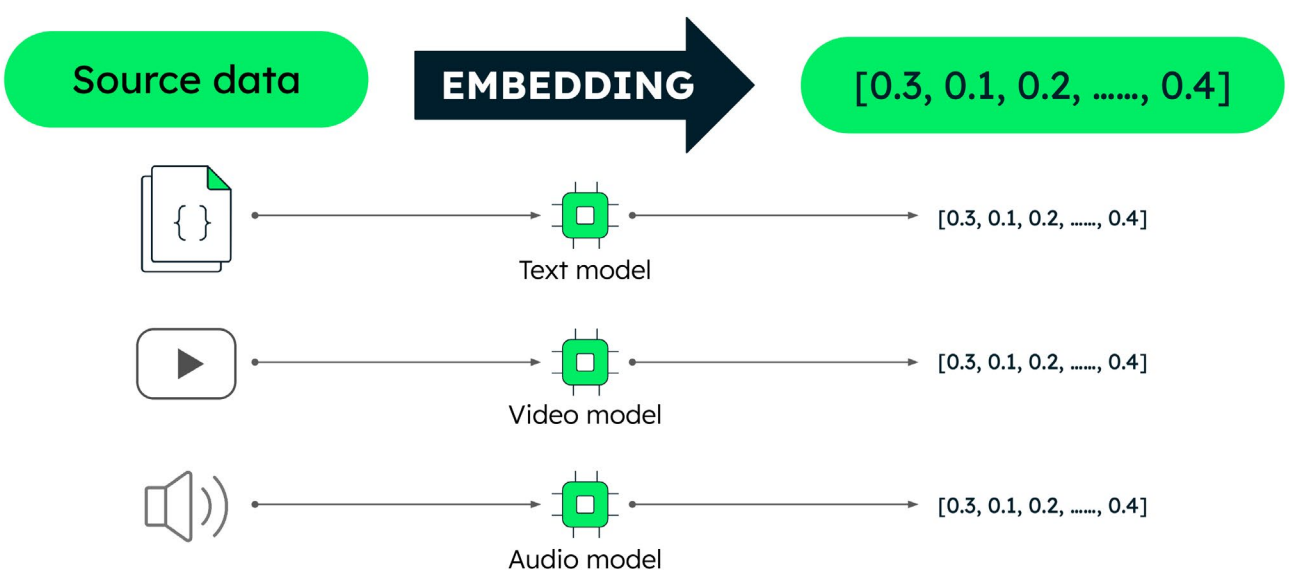

Figure 1. Vectors are a numeric representation of data and related context.

Vector embeddings can be searched to find similar content based on vectors being "near" one another in high-dimensional space. Vector embeddings represent enterprise data in a way that AI can comprehend, giving it context and characteristics, which the rows and columns of a relational database are unable to capture.

The database that stores vector embeddings is called a vector database, and the method for querying a vector database is called vector search (also known as semantic search). Vector search is the retrieval system responsible for finding relevant information from a knowledge base that is external to the LLM (Figure 2).

capable of capturing the time intervals between data records. And when organizations are in need of robust search capabilities, they look for ways to augment their tech stack with purpose-built search engines. As with these specific use cases, the tools you choose will have a huge impact on performance, scalability, and cost. There are a couple of paths you can choose from: You can bolt-on a standalone database, or you can take an integrated platform approach where the features you wish to add are supported natively by the database.

Standalone solutions (also called bolt-on, purpose-built, or niche) may sound like a quick fix for

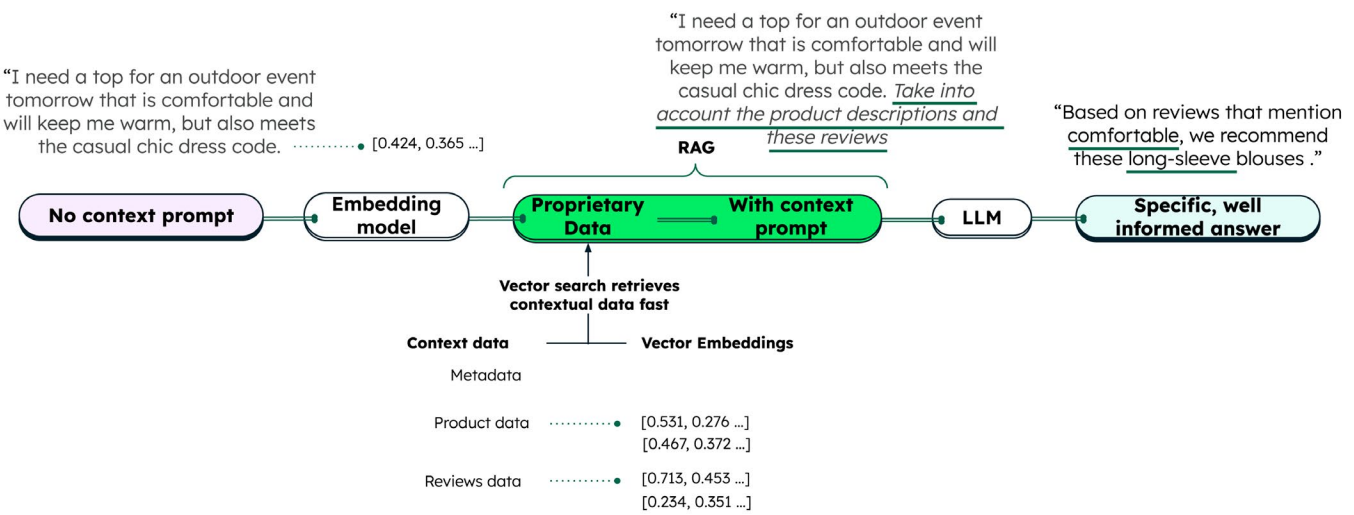

Figure 2. Augmenting a large language model (LLM) with RAG.

Taken together, these technology solutions and concepts form the foundation of AI tools and techniques. But how do you know which tools to leverage for your particular scenario? And crucially, how do you embrace AI functionality without increasing architectural sprawl and maintenance work, which often falls on the shoulders of developers?

Bolt-on and hang on

When it comes to onboarding AI solutions, in many ways, it's no different than niche use cases that came before. When mobile apps and IoT devices were first coming to market, they relied on a specific data type, time-series data, that is

tech problems, but they come with a bundle of challenges. From complex implementation and maintenance to domain specialization and data duplication, bolt-on tech can quickly create an unwieldy, expensive tech stack.

In today's macroeconomic environment, IT decision-makers are under pressure to do more with less. But that's not what bolt-on solutions are designed to do. As the term "tech stack" implies, doing more means adding more: a vector database here, a search engine there – time series, graph data, archive store, cache tier, the list goes on.

The functionality and capabilities may justify the upfront investment, but the increased cost of development combined with the ongoing

operational overhead of these complex solutions eventually catch up. Before you know it, you're replicating data across an increasingly complex architecture and hiring specialized consultants with expensive skill sets to operate and maintain an array of niche technology.

The following are all important considerations anytime you're considering augmenting the tech stack to add capabilities and features like vector or semantic search, full-text search, and time series databases:

Another database to learn – Each new programming language or API has a unique syntax, and often introduces new concepts that developers need to understand. This could include new ways of handling data, different types of data structures, or unique methods of querying data. Documentation and learning resources can accelerate the learning process but only if they're readily available, which isn't always the case.

Another database to sync – Large volumes of data can make synchronization processes slow and resource-intensive, and ensuring data consistency between different databases can be complex, especially in distributed systems. Changes often need to be captured in real time. System compatibility can also be an issue because different systems may have different formats and standards.

Another database to secure – Cyber threats are constantly evolving, and it's up to developers to stay informed and prepared through continuous learning and adaptation. They're also responsible for creating secure login systems, managing user roles and permissions, and handling password storage and recovery. Add to these responsibilities the important work of protecting sensitive data with strong encryption, managing encryption keys, and understanding security standards and regulations, all of which becomes more complicated with each new standalone technology you add to the stack.

Another database to integrate – Integrating a new database with your tech stack is easier said than done. It could involve writing new APIs, adjusting data models, or changing application logic. Ensuring data consistency across different systems is also crucial. Migrating existing data to the new database involves mapping data from the old schema to the new one, handling data transformation, and testing the migration process. In addition to all these tasks, developers need to ensure that the database performs well under different loads and that it can be reliably deployed in production.

Another database to scale – As applications grow or requirements change, it becomes increasingly difficult to access all necessary data at service levels that users expect. Data needs to be located where those users are, which means moving databases to branches and edge locations for low-latency performance. Developers also need to ensure that the database satisfies regulatory requirements. This involves maintaining multiple copies of data and automating recovery without adding excessive complexity. Scaling a database also often leads to escalating costs. Developers need to manage these costs while ensuring the database continues to perform optimally.

Another database to monitor – Developers are always on the hook for monitoring the database for performance issues, resource utilization, and potential security threats. Slow queries can impede application performance. This requires continuous monitoring and analysis of query execution times and usage of resources like CPU, memory, and storage. They also need to monitor for unusual patterns or spikes in database activity, which could indicate a problem or a potential security threat.

Another database to back up – Backing up a database involves planning, creating, verifying, and storing the backup in a secure and reliable location on-site, off-site, or in the cloud. Issues to consider include data consistency, backup storage size, and recovery time. It involves determining what data needs to be backed up, the frequency of backups, and where the backups will be stored. Finally, the recovery time (the time

it takes to restore the database from the backup) should be minimized to reduce downtime.

Developers already assume responsibility for many of these important tasks. For every standalone or purpose-built database an organization chooses to add to their tech stack, these responsibilities multiply in terms of the complexity of the task, the time it takes to do it, and the risk of not doing it right (Figure 3).

without humans having to intervene. Fraud detection is another common use case for fuzzy text search, where it's used to link transactions across accounts and identify suspicious transactions.

Search is used in auditing and discovery workloads. When organizations are handling sensitive information or data that's subject to a regulatory authority, they need the ability to audit

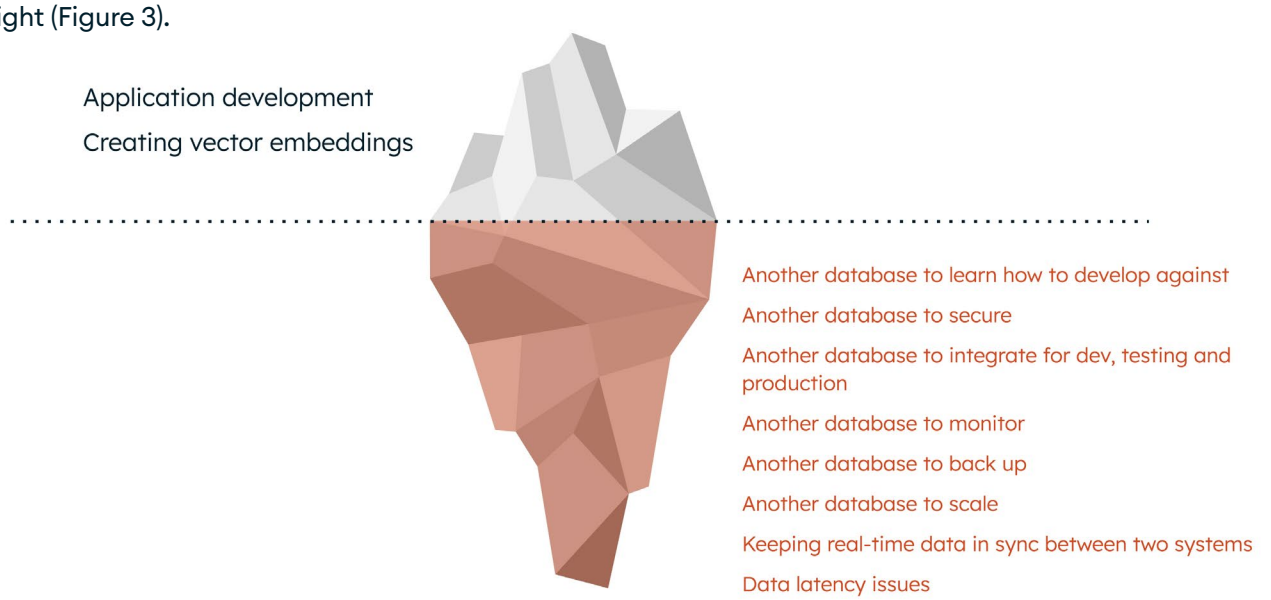

Figure 3. The hidden cost and complexity of bolting on a standalone database

Standalone search

When an application has been running for a number of years and generating an increasing amount of data, users often develop the need to search and filter data more efficiently. They may require the system to support query conditions and flexible data structures that go beyond what a standard database can provide. As user requirements evolve, demand often increases for must-have full-text search capabilities like auto-complete, faceted search, and pluralizations that can be stemmed back to root words.

Users today have become accustomed to full-text indexing that can support features like fuzzy search and synonyms so they can find results even when they don't know the exact phrase. Fuzzy search helps validate transactions on the backend

what's going on inside the system. Sometimes legal proceedings may require information about when an operation occurred, who initiated it, why, and at what time.

Standalone search solutions run alongside the main application database to provide features sought by users and customers (Figure 4). Adding a full-text search function on top of an operational database is one of the most common use cases. Popular search solutions include Apache Solr, Elasticsearch, Algolia, and OpenSearch.

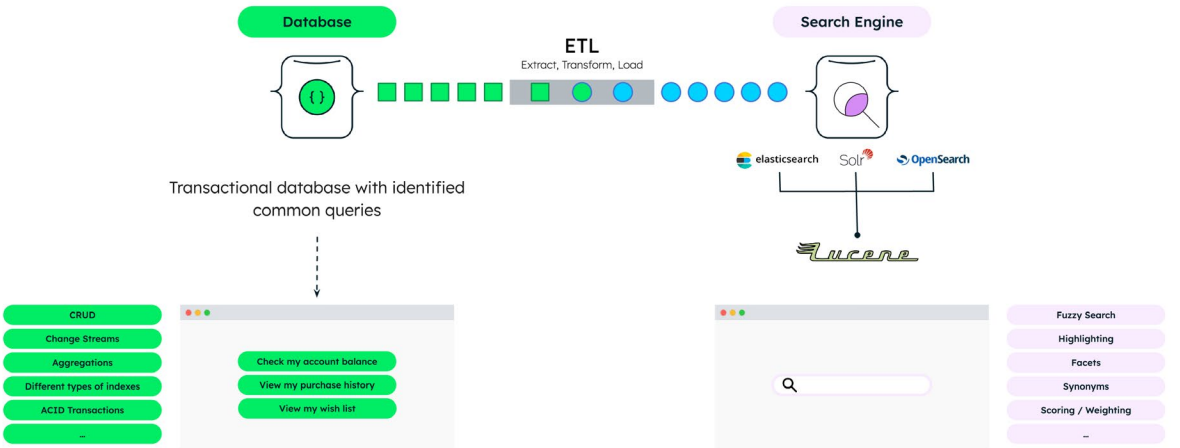

Figure 4. Traditional approach: a standalone search engine

Technology veterans have the ability to expertly deploy and optimize Elasticsearch, but attaining that level of expertise takes time and considerable trial and error. More often than not, it's the responsibility of one person or a small team figuring out how to stand up, configure, and optimize the new search environment as they go along.

First they'll need approval for and access to a virtual machine, containerization infrastructure, or a physical box to host the solution. The timeframe for getting permission to deploy virtual resources could be weeks, whereas purchasing physical hardware could take months.

The next step is right-sizing resources for compute, memory, and storage. Predicting the size of a workload is an inexact science. Overprovisioning helps ensure resources are available if there's a spike in activity, but it also leads to higher costs. So can guessing, which is all too common.

Standing up the search service also requires SSL certificates, security modules, authentication, and user permissions. For someone who's never done this before, it could take weeks to install and configure security and permissions for the service, all while potentially increasing the risk from common vulnerabilities and exposures (CVEs). That same person will most likely also be responsible for CVE monitoring and rapid response.

Next comes the complicated and consequential question of how to move data from the application to the search engine. To surface relevant and up-to-date search results, the database and search engine need a synchronization mechanism that replicates data from the database to the search engine. This typically involves a data pipeline with custom filtering and transformation logic built on top of messaging systems such as Apache Kafka, RabbitMQ, or using packaged connectors from specialized providers. Each requires developers to be well versed in connecting all the different services in order to expose the core functionality. They'll also need to be able to find errors when they arise, identify root causes, and remedy issues while balancing other responsibilities.

The synchronization mechanism has to be deployed onto its own nodes, creating additional hardware sprawl. The person or team responsible will have to solve for a list of setup issues:

> *What data should be moved into the search engine?*
> *What format should the data be brought over in?*
> *Should all the data be moved or just a portion?*
> *What tool am I going to use to bring the data over?*
> *Once a dataset has been moved, is there a way to capture just the newest data to avoid having to move all of it over?*
> *If you have to move it all over at once, when do you do it?*

Moving data between the application and search can be a massive undertaking. Copying data takes time, sometimes 24 hours or more for a terabyte of data. Syncing data in batches once a day directly impacts the user experience because any changes a user makes to a record won't show up in search for 24 hours or however long the interval is between syncs. This could result in outdated offerings on a website, a poor user experience, and a drag on revenue. The alternative is real-time syncing so that every time you write to the application, you write to the search environment as well. This entails learning client libraries and the different query and indexing languages.

Different databases have different data models, so it's important to align the data to the model of the target database. An ETL tool (extract, transform, load) will be required to translate the data from source to target. It's rarely a one-to-one connection between the source and target. If the source has been running for years, dozens of integrations may already exist. If there's a change to one table, figuring out all the implications of the change will require testing to make sure the change was handled properly.

This is a massive amount of work and responsibility, and a commensurate amount of risk that things will not go right. It's also a poor allocation of resources because most of it is non-differentiating work. Users expect robust search functionality so it's not a unique differentiator from a competitive perspective. It's a similar story when it comes to standalone time series databases for data in motion such as IoT sensors, where there are many data points for a single timestamp (i.e., high cardinality[1]).

Time series and streaming data

Another common use case for adding a standalone solution is time series data, which is commonly used in financial services, manufacturing, energy and utilities, healthcare, and internet of things (IoT). Time series data is processed chronologically in high volume and indexed or listed in time order. If you were to plot the points of time series data on a graph, one of your axes would always be time. The exponential rate of instrumentation and sensory data that continuously emits a stream of time series data is only set to grow given the increased emphasis on observability and measurement[2].

Time series data is very useful for identifying trends, such as whether a situation is improving or deteriorating. A few examples include high frequency trading applications for financial institutions, health monitoring of a patient's cerebral electrical activity or heart rate for healthcare institutions, and temperature measurements for building control or refrigeration units.

In IoT use cases, users track information using many data points from different sources. Examples include an agricultural customer with moisture sensors in their cornfield, a distribution center monitoring packages moving through its shipping and infrastructure network, or a manufacturer collecting data from sensors deployed on their assembly lines to track operational efficiency over time.

When building applications for these use cases, developers are presented with a dilemma: retrofit existing systems and face performance challenges or embrace specialized, bolt-on solutions with their unique set of complexities.

Retrofitting general purpose databases for time series data requires significant upfront work. Time-stamped data needs to be manually bucketed, and the right instrumentation set up. This eventually leads to inefficiencies in data storage, querying, analysis, and archiving, as general purpose databases are not inherently designed for time series data.

As the volume of data grows, the performance of these retrofitted solutions can degrade, leading to longer query times and scalability challenges. Bolt-on time series databases, while designed to handle time series data efficiently, are yet another solution to learn, manage, scale, secure, and procure. In addition, integrating time series

[1]An example of high cardinality – or many data points for a single timestamp – would be sensors on a car recording many different data points like location, objects nearby, location, etc.

[2]Accelerate Speed of Data and AI-Driven Business Value With Time Series Technology, Forbes January 2023

databases with the rest of the application and data architecture, ETL pipelines, and custom integrations adds fragility and complexity.

When working with streams of time series data, developers have to know different languages, APIs, drivers, and tools to bring streaming data (from sources like Apache Kafka) into their applications, increasing operational complexity with additional systems for teams to manage. Typically, developers process streams of data after landing events directly into their relational databases, which are not well suited for high volume, high velocity data. This often introduces latency into the system. It also makes for a poor developer experience because the rigid schemas of relational databases are difficult to adapt as user demands grow and functionality evolves. Working across streaming data and data in the database creates another friction point, driving up the cost of development since teams have to deal with new syntax and APIs. It also increases the risk of bad results or incorrect data being provided to business applications.

The worst part about all this complexity is that, like the extensive management tasks associated with managing a standalone search engine, it's non-differentiating work, meaning it doesn't set the business or the app apart from the competition. Users place a lot of trust that your app is delivering reliable information and won't tolerate inconsistencies. You can be happy if they reward you for a pleasant experience by returning, which isn't always the case, but they will definitely hold any friction they experience against you. So developers wind up spending time building DIY solutions for table-stakes functionality that they could be spending on differentiating features that will help the app gain traction among its users.

Lurking beneath the increased complexity and sprawl is the inherent risk involved when the knowledge required to operate different parts of the stack is spread thinly across different members of the team. In fact, having more systems that require deep expertise increases the likelihood that you'll be using at least one of them incorrectly.

Having multiple databases in your tech stack of course requires additional training for different systems, query languages, workflows, and APIs. In today's highly fluid workforce, there's always the risk that, once team members are trained on the relevant systems, they'll leave for other opportunities and need to be replaced with new personnel, who will then need to be trained in multiple systems.

Developer experience

It may be obvious by now, but it's often the developer who's responsible for acquiring the domain expertise necessary for setting up, scaling, and maintaining standalone databases. Having and keeping the expertise to maintain these systems on an ongoing basis is one of the biggest drivers of escalating cloud costs.

Standing up and maintaining a separate search engine alongside the database means developers will have to learn two different query languages to access the database and the search engine. This increases the learning curve for developers and forces frequent context switching when building. It also complicates testing and ongoing maintenance.

All the different sync tools, translation layers, message buses, and ETL code make it hard to anticipate disruptions. Imagine someone changes a schema format without notifying other team members and it results in a nightmare chain reaction: A change-stream interaction fails and breaks full-text indexing. That pipeline is throwing exceptions on the stream processor and not catching it. Now you need to fully rebuild the index. But when did it break? And by how much? How do you record what happened in the meantime? If it's been more than 24 hours since the last change-stream, it's already rolled over, so you can't just reprocess it. Essentially, you have to rewrite all the full-text indexes, and it's all happening during an unplanned maintenance window, so services are down. If the table has a lot of data – say terabytes of data – it could be hours before you restore full text indexing.

Service interruptions are often caused by brittle pipelines. When a developer introduces a new feature to an application, it necessitates changes to the database's schema that will affect the synchronization logic and search engine schema. This creates more dependencies that slow down the pace of rolling new features into production. These are the real real costs of purpose-built search solutions.

The story is much the same with mobile and IoT, where the undifferentiated heavy-lifting comes in the form of building and managing bidirectional sync between the mobile database, caching layers, ETLs, and the backend (Figure 5). The first step is figuring out what subset of data you actually need to sync to the mobile device, and whether you're going to only authorize this particular user to sync this data with this device. This involves authorization code that handles the question of what data is actually going to be available on this device. It also requires code that determines if the user has permission to update that data. Then the developer has to decide, out of all the data that users can download to their devices, what subset of that data will be downloadable to multiple users? And out of all the users, who is able to make simultaneous and conflicting updates? And can they sync that data with the backend and each other directly? Conflict-resolution code is necessary to decide who wins if two users update the same thing.

Networking is also a key decision when building apps for mobile and IoT. Will devices automatically connect to the backend in the background or do users have to take action? Is there enough processing power and storage to handle periodic demands for data? What happens when the network connection fails halfway through? Will users on the front end and the database on the backend all remain in a consistent and usable state even if sync happens to fail?

It's up to the developer to provide answers to all of these questions through meaningful design. These answers must have a test plan written for them, they have to be coded, and they have to go through QA and support to make sure the application achieves the design goals. It has to be performant because app users have high expectations. And all these decisions require domain knowledge to configure, manage, and update on an ongoing basis.

The challenge in IT today is the same as it has always been. How can additional functionality be introduced to support evolving workload requirements while avoiding architecture sprawl and increasing cost?

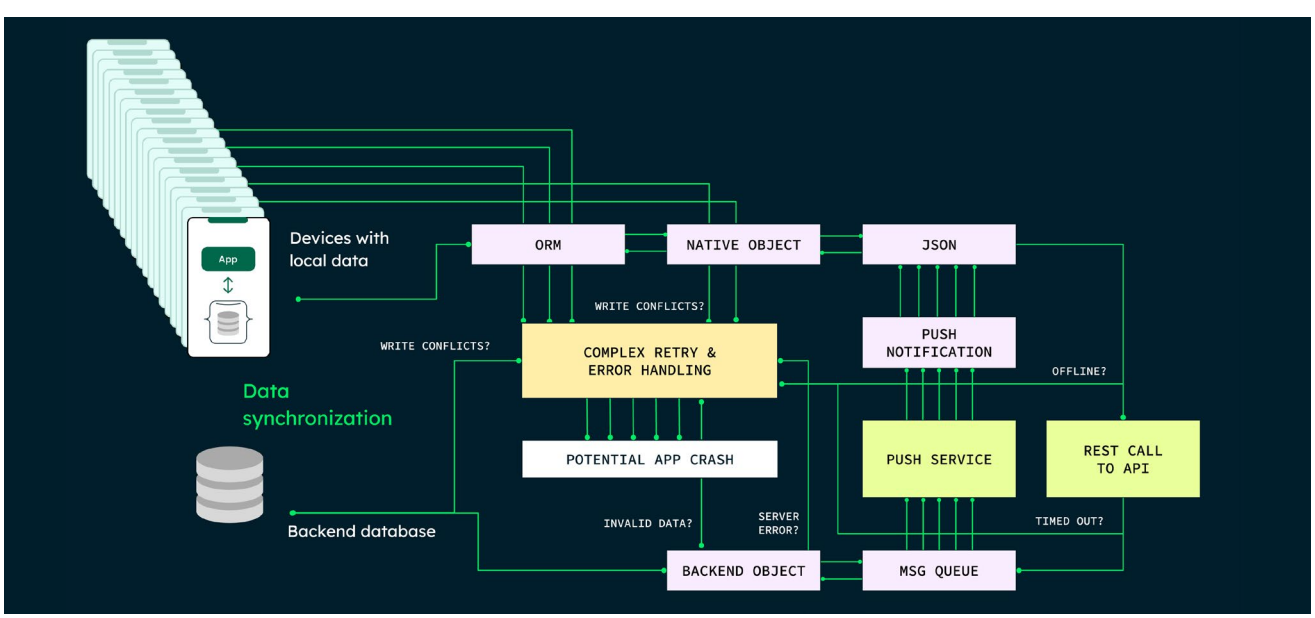

Figure 5. Real-time experiences at scale require complex conflict resolution.

The modern data platform

At MongoDB, we've been singing the praises of the document model since our inception.

The document model is the most intuitive, performant, and agile way to store application data. Documents let you store objects in the same format that you use in your application. You can embed related data instead of splitting it across multiple tables. This means better read performance, as you can minimize the number of places your operations are accessing on disk. And, since the structure of each document is flexible, you can easily evolve your data model throughout development. This flexibility makes document databases like MongoDB Atlas ideal for handling multiple different data types and workloads without having to onboard specialized, standalone solutions like those described above.

With Atlas Search, you can iterate and create multiple indexes that may or may not work, or where part of your application – most likely the legacy application – consumes it. You can then continue to work on the backend to improve the user experience by creating a second index. When the index is working properly, you can transition the application over to the new index. It doesn't cost any extra time, sync processes, or cognitive load. If you have a collection, you just create an index on top of it. If it isn't right, you simply create another index on top of it and delete the first. If that one isn't right, you add some fields to it, and now you can query it – after about a half-day's worth of experimentation.

Atlas Search runs alongside the Apache Lucene search index. There's no separate system to operate and maintain, and everything is fully synchronized and managed. The database and search engine are highly available through a distributed architecture with self-healing recovery, relieving developers from tons of non-differentiating work (Figure 6).

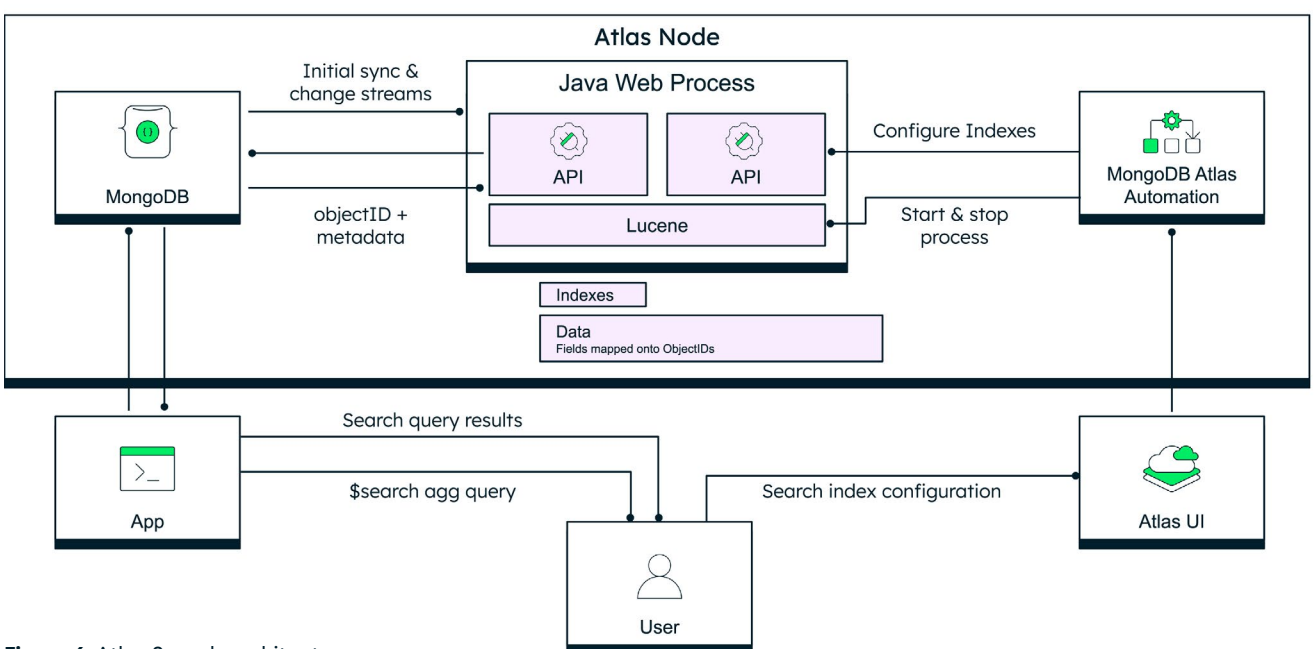

Figure 6. Atlas Search architecture

Atlas eliminates the need to replicate chunks of data from a time series database into a NoSQL database, or from NoSQL into an object store. The data sits in one repository that can be accessed by multiple services. If a developer needs to integrate time series data, they simply create a native time series collection in a new Atlas cluster or an existing one. The developer has one API to deal with and there's no need to set up a time series data stream or learn a new time-series API.

The hardest part of working with data at the edge of the network – syncing with the backend – is entirely streamlined through an integrated feature in Atlas called Device Sync, which handles sync, networking, and conflict resolution, eliminating the cost and complexity of DIY solutions. It solves bidirectional sync with cascading rules and permissions built on the backend. Users can create and update rules to establish read and write permissions on each document and each field within that document using a user-friendly GUI or a sophisticated command line interface (CLI).

Device Sync allows the user to generate a set of sample data with a schema on Atlas and then use that to generate code that can be used for the mobile database. End users can continue using an app even if the device goes offline. Then, when the device comes back online, Atlas Device Sync will automatically reconnect and resynchronize changes between the front and back ends.

When it comes to processing streaming data, Atlas Stream Processing simplifies building and monitoring modern event-driven applications by removing the need to introduce new or specialized infrastructure components required for stream processing. It increases developer productivity by enabling teams to react to and extract insights from high volumes of streaming data. The document data model is uniquely capable of meeting the needs of rich, complex streams of data, reducing friction related to rigid schemas and enabling adaptability as your application grows. Atlas Stream Processing reduces the amount of code a developer has to write to process streaming data, making it faster to iterate, easier to validate for data correctness, and simpler to gain visibility, all with less moving parts to manage.

The emergence of AI-powered applications and use cases only serves to confirm the flexibility and versatility of the document model, which is the foundation of the MongoDB Atlas developer data platform. The primary "intelligence" behind AI is reams of unstructured data that is used to train machine learning models. From video and call transcripts to documentation and transaction histories, organizations can leverage their own proprietary business data to create truly unique AI-powered applications and customer experiences. And no database is better suited to handle vast quantities of unstructured data than a document database. MongoDB was designed for managing humongous (hence the "mongo" in MongoDB) amounts of documents. And our retrieval and storage mechanisms were built to optimize for AI-type workload patterns. No other database can simply bolt on this type of functionality.

With standalone databases, the core object store, search, and archive functionality all exist in different services that have to be connected. In Atlas, there's no glue code or schema to write because it's already built in. You query the API and it fetches the data. If it's there locally, it fetches it. If it's in an online archive, it fetches it. You don't have to translate it. That's an enormous amount of work developers don't have to do.

MongoDB Atlas and Atlas Vector Search enable organizations to store vector embeddings right alongside their other operational data. And developers can query and index that data without having to execute expensive and time-consuming ETL operations. By storing vectors together with operational data, you avoid the need to sync data between your application database and your vector store at both query and write time. It also simplifies the overall architecture of your application. You don't need to maintain a separate service or database for the vectors, reducing the complexity and potential points of failure in your system. MongoDB Atlas with vector search also scales horizontally and vertically, allowing you to power the most demanding workloads.

With the introduction of Atlas Search Nodes, you can deploy dedicated infrastructure for Atlas Search and Vector Search workloads, allowing you to fully scale search independent of database needs. Incorporating Search Nodes into your Atlas deployment allows for better performance at scale and delivers workload isolation, higher availability, and the ability to optimize resource usage.

MongoDB is integrated with a rich ecosystem of AI developer frameworks, LLMs, and embedding providers. This, combined with our industry-leading multi-cloud capabilities, allows organizations flexibility to move quickly and avoid lock-in to any particular AI technology in a fast-moving space.

Built-in or bolt-on?

While specialized standalone or bolt-on solutions provide users with the rich experiences they expect, the ongoing maintenance and specialized domain expertise required outweigh the initial business justification. The application stack gets more complex and unwieldy, which translates to reduced developer velocity, compromised customer experience, and escalating costs. An integrated, modern data platform like MongoDB Atlas comes with built-in tools for working with all different types of data, data models, and use cases. It enables developers to meet the application requirements they face today. Atlas provides the maturity of relational databases at the performance of a modern general-purpose data platform. All Atlas capabilities are seamlessly integrated, reducing much of the infrastructure and data duplication, maintenance, developer overhead and the much of the hidden costs associated with bolt-on solutions (Figure 7).

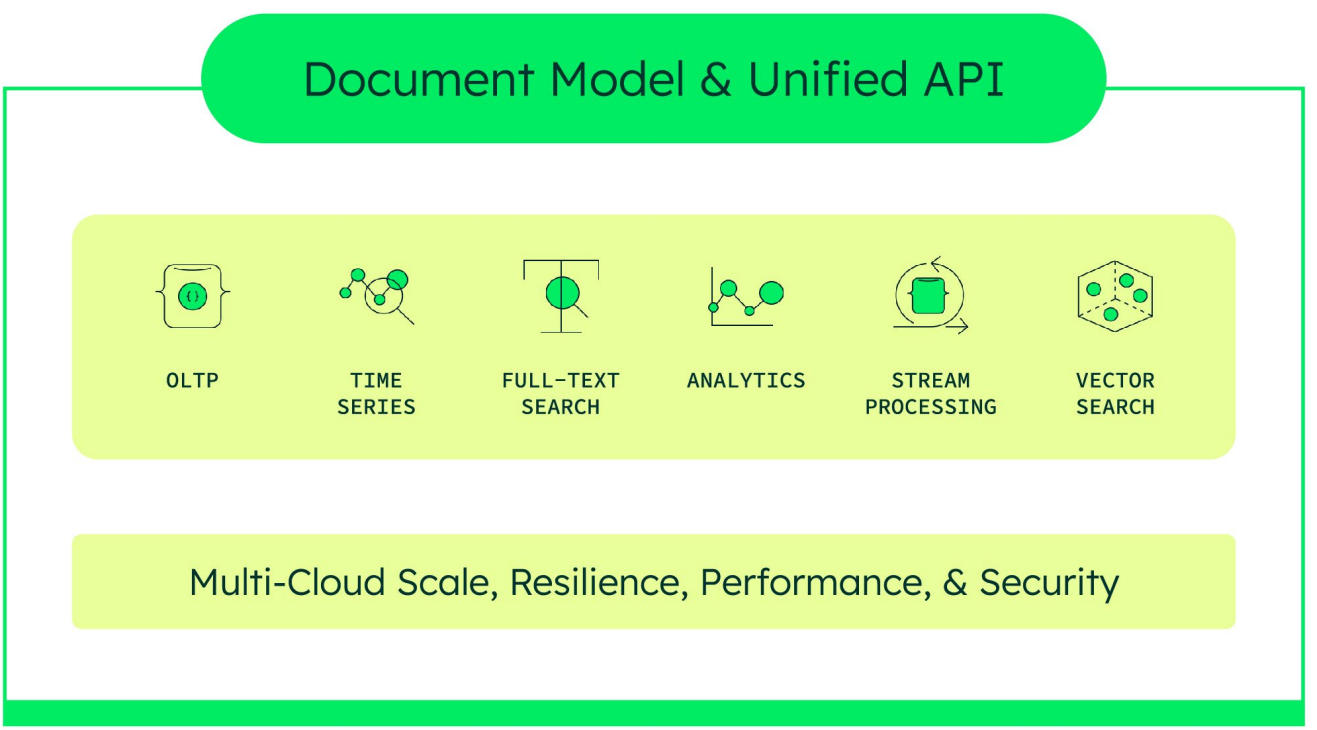

Figure 7. A modern data platform gives you integrated services rather than bolting on a standalone solution.

Getting started with Atlas

MongoDB Atlas offers a forever-free tier for development. Once deployed, it only takes a few clicks of a button to add functionality to your application. To learn more about using Atlas for AI use cases, visit our Atlas Vector Search page, or check out our vector search tutorial. To get started with full-text search, read our getting started tutorial. You can also read our Atlas Search documentation, which provides a complete reference on how to configure, manage, and query search indexes, along with performance recommendations. To find out how to use Atlas Search Nodes to provide dedicated infrastructure for Atlas Search and Vector Search workloads, read our Search Nodes documentation. Learn how MongoDB Atlas supports time series collections. To leverage Atlas for streaming data use cases, read our tutorial on getting started with Atlas Stream Processing. To learn more about data-in-motion use cases, visit our Atlas Device Sync product page.