The Fading Era of Bespoke Databases: Multi-Model Trends and Implications (2026–2030)

Executive Summary

Over the next decade, single-purpose “bespoke” databases are expected to lose prominence as multi-model, general-purpose data platforms rise to dominance. **Multi-model databases** – systems supporting multiple data models or workloads under one engine – are becoming mainstream, reducing the need for a separate specialty database for every use case. The **majority of new database systems launched from 2026 onward are likely to be multi-model** rather than narrowly focused, aiming to handle relational, document, graph, key–value, and even analytical workloads within one platform. This consolidation trend is driven by user demands for simpler data architectures and lower Total Cost of Ownership (TCO). Organizations adopting multi-model databases report **infrastructure consolidation, operational efficiency gains, and faster development cycles** . In practice, this means fewer siloed databases: instead of maintaining one engine for transactions, another for analytics, and others for documents or time-series, companies can increasingly leverage unified solutions. For example, SingleStoreDB (formerly MemSQL) illustrates this by combining relational storage with native support for time-series, JSON, and geospatial data in one engine . Such platforms promise to *“eliminate the complexity of traditional polyglot persistence”* by enabling multiple data models and queries in one place .

Crucially, **specialized databases for time-series and vector data are being absorbed into general-purpose engines.** The explosion of purpose-built time-series and vector DBMSs in the early 2020s (e.g. InfluxDB, Pinecone) is giving way to their functionality becoming *features* of mainstream databases. The open-source relational stalwarts have rapidly evolved to incorporate these new data types: PostgreSQL can be extended with TimescaleDB for time-series and with the pgvector module for AI embeddings, while MySQL’s HeatWave engine now includes an in-database vector store . These additions indicate that *standalone vector and time-series databases may not remain separate categories for long*. In fact, by 2025 many developers were opting to implement vector search using existing systems (Postgres + pgvector, MongoDB Atlas Vector, or Redis) rather than adopting a dedicated vector DB – a consolidated approach that became “the norm” . A recent industry survey showed that PostgreSQL (with pgvector) and MongoDB had **double the usage share of specialized vector databases like Pinecone or Chroma** . Developers prefer extending familiar databases over introducing new ones, and these generalist platforms have proven capable: *Postgres with pgvector can achieve performance competitive with purpose-built vector stores* , and TimescaleDB (a Postgres extension) can even outperform InfluxDB on certain high-cardinality

time-series workloads . As a result, **the performance gap between generalist databases and niche engines is narrowing**. Innovations like hybrid transactional/analytical processing (HTAP) and improved indexing have boosted general-purpose databases on workloads once dominated by specialized systems. Oracle's MySQL HeatWave, for example, adds a columnar analytics module and machine learning inside MySQL itself, delivering analytics speedups on par with dedicated data warehouses . These trends suggest that by 2030, a well-tuned multi-model database will often meet "80/20" requirements across different use cases – eroding the justification for maintaining many bespoke DBMS technologies.

In the cloud era, **database service providers are emphasizing API compatibility over proprietary engines**, accelerating the decline of bespoke platforms. All major cloud vendors now offer database services that are branded by compatibility: e.g. *Amazon Aurora is explicitly marketed as "MySQL- and PostgreSQL-compatible"*, capable of running existing MySQL/Postgres applications without changes . AWS's portfolio includes services like Amazon DocumentDB **"with MongoDB compatibility"** and Amazon Keyspaces **"for Apache Cassandra"**, underscoring that what they sell is the interface, not a new query dialect . Similarly, Azure Cosmos DB provides MongoDB and Cassandra API options, and Google Cloud Spanner offers a PostgreSQL interface – all to lower adoption friction by piggybacking on popular standards. **Customers increasingly care that a service supports a familiar API or pattern, rather than the engine's internal name.** This is reflected in procurement: technical decision-makers and RFPs are now more likely to specify required capabilities (e.g. "must be PostgreSQL-compatible" or "must support GraphQL queries" etc.) instead of mandating a particular vendor product. For example, HashiCorp's Terraform Enterprise documentation explicitly allows *"a PostgreSQL-compatible server such as Amazon Aurora PostgreSQL"* to satisfy its database requirement – indicating that compatibility with Postgres is viewed as equivalent to Postgres itself. By focusing on standard interfaces, organizations preserve flexibility to swap underlying technologies. In practice, a cloud buyer may ask for *"a MySQL-compatible, highly-scalable database service"*, inviting multiple providers to bid solutions that meet that interface, rather than specifically requesting "Aurora" or "MySQL Enterprise". This shift empowers users to avoid lock-in and favors those general-purpose engines that have broad API compatibility.

Another facet of this evolution is the prevalence of **drop-in replacements and protocol-compatible alternatives** supplanting the original forked databases. In the 2010s, open-source databases often saw community forks (e.g. MariaDB from MySQL) to add features or license flexibility. But in the 2020s, an alternate pattern emerged: *new engines reimplemented popular DBMS protocols from scratch*, delivering improved performance or cloud-native designs while remaining a seamless substitute. Amazon Aurora was an early exemplar – a cloud-native storage engine built to be a *fully compatible MySQL 5.6 substitute with ~5× higher throughput* . Aurora quickly gained traction on AWS, to the point that MySQL's market share statistics often exclude Aurora and other MySQL-compatible variants . Similarly, ScyllaDB was created as a C++ rewrite of Apache Cassandra, offering an order-of-magnitude throughput boost while speaking Cassandra's query language and protocols . The success of these platforms demonstrated that organizations are willing to "swap out" the internals of their database if it yields better performance or lower cost – *as long as the external interface remains the same*.

This has happened repeatedly: many MySQL users have transitioned to MariaDB or Percona Server (enhanced MySQL forks) or to cloud offerings like Aurora, without application changes . MongoDB's restrictive licensing, likewise, spurred the rise of protocol-compatible substitutes – Amazon's DocumentDB and the open-source FerretDB both implement the MongoDB API on different underlying engines (PostgreSQL in FerretDB's case) . Even the event streaming world saw Redpanda emerge as a drop-in Kafka replacement (no ZooKeeper required). **The frequency of such migrations to compatible alternatives has increased**, as evidenced by MySQL's ecosystem: an estimated 20% of MySQL's market presence comes from compatible forks or cloud reimplementations not counted in "official" MySQL numbers . In short, whenever a incumbent database hits limits – technical or licensing – users now look for a plug-compatible replacement rather than an entirely different API. This further undermines the "bespoke" databases: if a specialized system stagnates, its user base can be captured by a more general platform that simply offers a compatible interface with better characteristics.

Overall, by 2026–2030 we expect **multi-model, general-purpose databases to dominate new deployments**, while narrowly specialized databases see reduced adoption except in extreme edge cases. Cloud-first deployment models and the need for agility are reinforcing this convergence. As Gartner notes, already *61% of the database market revenue is cloud-based and 91% of new database growth is in the cloud* – an arena where managed services that provide familiar interfaces thrive. The lines between traditional categories (relational vs NoSQL, OLTP vs OLAP) are blurring: modern systems either natively support mixed workloads or tightly integrate with complementary tools. Decision-makers will prioritize solutions that minimize complexity: if a single cloud database service can satisfy multiple requirements (transactional, analytical, semi-structured, etc.) with acceptable performance, it will often be chosen over maintaining a constellation of bespoke databases. This report examines each aspect of this trend in detail – from the rise of multi-model databases and the absorption of vectors/time-series into mainstream DBMS, to the cloud vendors' focus on compatibility, the role of drop-in replacements, performance considerations, and changing procurement behaviors – and also considers counter-evidence and scenarios where specialized databases might persist.

(Next, we provide a detailed analysis with supporting evidence, followed by a section on counter-evidence challenging the "bespoke DB era fades" narrative.)

Introduction: From Polyglot Persistence to Multi-Model Unification

In the 2010s, the database landscape exploded into a menagerie of specialized systems, each tailored to a particular data model or workload. The credo of "polyglot persistence" – using the right database for each job – led to architectures with separate engines for relational data, documents, graphs, time-series, search indexes, analytics, and more. This *bespoke database era* delivered technical optimizations for each data type, but at the cost of significant complexity in operations and development. Every additional database brought its own query language, scaling strategy, consistency quirks, and maintenance overhead. By the early 2020s,

organizations began feeling the pain of this sprawl: having five different DBMS products in production might mean five different backup/HA setups, data duplication between systems, and scarce expertise in each niche technology. As a result, a pendulum swing back toward consolidation is now underway. The industry is asking: *can a smaller set of powerful, multi-model databases satisfy most needs, obviating the dozens of one-trick systems?*

Recent technological advances suggest the answer is “yes” in many cases. Major relational databases, once limited to strictly structured tabular data, have evolved dramatically to embrace flexibility. PostgreSQL and MySQL, for instance, added native JSON document types, XML, key–value extensions, and more over the past decade . They incorporated horizontal scaling capabilities (via plugins or cloud-native variants) to handle larger volumes, and even started supporting semi-structured and unstructured data that previously required NoSQL stores . At the same time, some NoSQL platforms added SQL-like querying and transaction options to broaden their use (e.g. MongoDB’s ACID transactions and SQL interfaces in newer versions). The consequence is a convergence: *from the application developer’s perspective, the gap between a “relational” and “NoSQL” database is much less stark than it was a decade ago* . Gartner foresaw this trend in the mid-2010s, predicting that leading vendors would offer multiple data models (relational + NoSQL) in one system . While technical purists debated the feasibility, the market direction has indeed been toward multi-model integration.

Today (mid-2020s), multi-model databases are no longer hype but reality in many deployments. A multi-model DBMS is one that **supports diverse data models (document, graph, key–value, relational, etc.) through a single engine and interface** . The goal is to achieve the benefits of polyglot persistence (flexible modeling for different data) without its biggest drawback (operational complexity of many systems) . Early exemplars included ArangoDB (with JSON and graph modes) and OrientDB, but now even the giants like Oracle, Microsoft, and IBM have positioned their flagship databases as multi-model (for example, Oracle Database supports JSON, XML, spatial, graph inside the same engine). Cloud data services further accelerate this, as they can hide engine complexities behind a unified API – e.g. Azure Cosmos DB offers multiple APIs (SQL, MongoDB, Cassandra, Gremlin graph) on a single backend. The **market impact** of multi-model databases has been significant: companies report being able to consolidate workloads that previously required 2–3 different databases into one platform . This consolidation yields **lower infrastructure costs and faster development**, since teams can work with one system and one query language across use cases . Developers benefit from not having to learn and integrate multiple query languages (SQL, plus maybe Gremlin, plus a proprietary time-series query, etc.), which improves productivity and reduces errors. Business leaders are drawn to the **TCO reduction** – fewer licenses/support contracts and simpler operations. These advantages are driving multi-model adoption in sectors from e-commerce (combining product catalogs, user graphs, and transactions) to IoT (time-series telemetry plus metadata) .

To be clear, the multi-model approach is not a silver bullet, and polyglot persistence is not dead yet. Some experts remain skeptical that any single database can be “best” at everything – warning that a jack-of-all-trades system might be master of none . Indeed, historically, attempts to force all data into one model (e.g. shoehorning hierarchical or graph data into relational tables

in the 2000s) led to suboptimal outcomes, which is what gave rise to specialized NoSQL stores in the first place . The key difference now is that modern multi-model databases aren't *forcing* disparate data into one rigid model; instead, they natively support multiple models with appropriate storage/indexing under the hood. In essence, they bundle what would have been multiple engines into one product – ideally sharing common infrastructure (security, clustering, etc.) while optimizing each model's performance internally. This approach can succeed as long as the trade-offs are well-managed. In practice, many multi-model DBMS use *pluggable storage engines or modules* for each data type (for example, a JSON document might be stored using a different internal structure than a relational table in the same DB). This modularity aims to preserve much of the performance specialization. Nonetheless, the **performance and complexity trade-offs** of multi-model systems require scrutiny. In this report, we will investigate if generalist databases can truly match specialists on their home turf (performance gap analysis), and we'll review evidence both for and against the notion that the "bespoke DB era" is ending.

The sections that follow address specific facets of this broad trend:

- **Multi-Model vs. Single-Purpose Databases: 2026–2030 Outlook.** We examine the expected share of new databases that are multi-model versus specialized single-model systems in the late 2020s, based on current product roadmaps and industry sentiment.

- **Absorption of Time-Series and Vector Databases into General Engines.** A focused look at two prominent specialist categories – time-series databases and vector similarity search databases – and whether their functionality is being subsumed by general platforms.

- **Cloud Databases: API Compatibility Over Engine Brand.** How cloud providers market and deliver database services, and the shift toward emphasizing wire-protocol or API compatibility (PostgreSQL, MySQL, MongoDB, etc.) rather than introducing entirely new proprietary interfaces.

- **Forks vs. Drop-In Replacements: Protocol Compatibility in Practice.** The phenomenon of organizations replacing one database with another that "speaks the same language," including examples like Aurora/MySQL, MariaDB, Scylla/Cassandra, and what this implies for database lifecycles.

- **Performance Gap Narrowing: Generalists vs. Specialists.** Analysis of whether multi-model or general-purpose databases can approach the performance of purpose-built systems for key workloads (analytical queries, ingest rates, etc.), citing benchmarks and vendor innovations (like HTAP, in-memory, vector indexing).

- **Procurement Patterns: RFPs Specifying Interfaces/Patterns Instead of Products.** Observations on how enterprise procurement and RFP language is evolving – focusing on required capabilities (e.g. "must support graph queries" or "must be PostgreSQL-compatible") rather than mandating a specific database product by name,

reflecting the commoditization of the database interface.

We will then present a **Counter-Evidence** section discussing scenarios and arguments where specialized databases might *not* fade – for example, cases where a one-size-fits-all approach falls short, or where the proliferation of new data types spawns *new* specialist categories even as older ones consolidate. Finally, we conclude with an integrated perspective on the future database landscape for technical decision-makers.

Multi-Model vs. Single-Purpose: New Database Trends in the Late 2020s

The database industry's innovation engine is now firmly pointed toward multi-model and multi-workload systems. When surveying the "new generation" of database technologies (those emerging or gaining prominence in the mid-to-late 2020s), a clear pattern emerges: **most new database platforms are being designed with versatility in mind, rather than as single-purpose tools.** In other words, the era when each new data problem sparked an entirely new type of database (as seen in the 2010–2015 NoSQL boom) has given way to an era where new databases aim to cover multiple problems at once. Industry analysts and publications increasingly highlight multi-model capabilities as a key selling point of modern databases. In 2022, a DB-Engines article noted that *"Relational is as strong today as ever"* precisely because relational vendors extended their products to meet new requirements – ushering in *"the era of multi-model database systems"* . The article specifically cites that adding JSON, time-series, and other features to relational engines has been a successful strategy to keep them dominant . This reflects a broader truth: rather than building entirely new bespoke DBMS for each data type, the market is rewarding databases that **incorporate new data types into an existing, proven core** (often the relational core).

Share of new DBMS that are multi-model (2026–2030): While it's impossible to precisely quantify "share of new databases" in advance, all indicators suggest that *the majority of databases developed or heavily promoted in the 2026–2030 timeframe will be multi-model or at least multi-workload.* Many recent entrants in the database arena advertise either multi-model support or convergence of transactional and analytical capabilities (HTAP). For instance, Google's AlloyDB (launched 2022) is built on PostgreSQL and pitched for both analytics and transactions (mixing workloads). Snowflake, while primarily analytical, is adding support for transactional use cases (e.g. Unistore) – effectively moving towards a multi-workload system. On the operational side, distributed SQL databases like CockroachDB and YugabyteDB use PostgreSQL compatibility and aim to handle both OLTP and some analytical querying, as well as multi-model data (Yugabyte has a key–value API alongside SQL). The **emphasis on flexibility** is also evident in startup messaging: database startups founded in recent years (e.g. FaunaDB, TileDB, HarperDB) often tout their ability to support multiple paradigms or query types out of the box.

Meanwhile, incumbent vendors are repositioning existing products as multi-model rather than introducing separate new engines. For example, Microsoft's Azure SQL Database now supports graph queries and JSON, and Microsoft is even building a document database (MongoDB-compatible) *on top of PostgreSQL* rather than creating one from scratch . This approach suggests that the creation of entirely new single-purpose engines is slowing; instead, new solutions are frequently implemented as layers on an existing engine (as with PostgreSQL becoming a platform for time-series, document, and now MongoDB-compatible store via extensions). The result is fewer net-new engine codebases and more feature extensions. Thus, by the late 2020s, we anticipate a **smaller number of generalized engines capturing a large share of new deployments**, each with plugin-like modules for various models, rather than a proliferation of brand-new specialist DBMS.

Some quantitative insight comes from market research and expert commentary. The *Expert Group for Database Management Systems* has identified *"the adoption of multi-model databases"* as a key trend driving the DBMS market growth through 2030 . They note that leading vendors (Oracle, AWS, etc.) *"focus on multi-model databases for disparate data sources"*, implying that multi-model capabilities are seen as essential for capturing new use cases . Even traditionally specialist NoSQL vendors like MongoDB have expanded beyond a single model – MongoDB added support for graph-like $lookup (joins) and time-series collections, and its Atlas platform now includes full-text search and data lake features. In effect, each "NoSQL" is becoming more multi-model too. A 2025 industry outlook from Rapydo observed that *"unified platforms that can handle diverse workloads"* are in demand . The same source predicts that by 2035, MySQL and PostgreSQL will have evolved to be *"multi-model to a high degree,"* possibly incorporating graph and time-series as first-class features .

There is also the notion of **multi-model vs. polyglot persistence** in architectural best practices. A panel of database experts on InfoQ debated this topic: some argued that using a small set of specialized databases (polyglot) is still wise, but others pointed out the operational burden and leaned towards multi-model solutions for newer projects . Notably, one expert stated *"in the future all of the major NoSQL databases will tend to share features of one another's native models"* , essentially forecasting convergence. Another expert in that panel noted that while Gartner was bullish on multi-model "one size fits all" platforms, in practice some leading vendors weren't fully there yet as of that time . However, by 2025, we see that many of those gaps have closed: Oracle, Microsoft, AWS all have multi-model offerings (Oracle's converged database, Cosmos DB's multi-model API, etc.).

The **benefits driving multi-model adoption** include reduced data duplication (one system can expose the same data in multiple ways), simpler development (one system to learn), and more **resilient data infrastructure** (fewer moving parts means fewer points of failure) . Organizations that *"master multi-model architectures gain competitive advantages through reduced complexity [and] improved developer productivity"* according to analysis in 2025 . These advantages are hard for single-purpose new entrants to compete with, unless a workload truly cannot be handled by any existing generalist. Thus, entrepreneurs and open-source communities launching new DB projects are incentivized to either extend an existing engine (as a plugin or

fork) or create something with broad applicability (to attract a wide user base), rather than a narrow specialist.

In summary, **from 2026 to 2030, we expect multi-model databases to represent a growing majority of new database adoption**, especially in enterprise and cloud environments. Single-purpose databases will continue to exist, but many will find themselves either: (a) subsumed as features within larger platforms, (b) relegated to niche uses, or (c) outpaced by multi-model competitors. The term "NoSQL" itself is becoming less relevant as multi-model databases blur the old distinctions – Rapydo suggests the rise of multi-model DBMS could *"make the term 'NoSQL' obsolete"*, since modern relational systems now handle JSON and other once-NoSQL features . In the late 2020s, a CIO or CTO evaluating database choices is more likely to compare **service capabilities (scalability, consistency, compatibility)** than to focus on the underlying data model category. The question has shifted from "Which kind of database (relational, graph, etc.) do I choose for this data?" to "Can I find one database solution that meets all these data needs adequately?" – and increasingly, the answer is yes.

Absorption of Time-Series and Vector Databases into General Engines

Two prominent categories of "bespoke" databases that saw rapid growth in the 2015–2023 period are **time-series databases** and **vector databases**. Time-series databases (TSDBs) like InfluxDB, TimescaleDB, OpenTSDB, and Graphite were optimized for storing sequences of measurements over time (IoT readings, metrics, logs, etc.), often offering high write throughput and compression for append-only data. Vector databases, which gained fame around 2023 with the rise of AI applications, are designed for storing high-dimensional vectors (embeddings) and performing similarity search for AI/ML use cases (examples include Pinecone, Weaviate, Milvus). Both categories initially emerged because general-purpose databases were lacking in functionality or efficiency for these specialized needs. However, we are now witnessing a **convergence where general engines absorb these specialized capabilities, diminishing the need for standalone TSDBs or vector DBs** in many scenarios.

Time-Series Databases absorption: The time-series data type has been a prime candidate for integration into mainstream databases. PostgreSQL provides a telling case study. Rather than develop a completely separate product, an innovative approach came via TimescaleDB – which is packaged as a PostgreSQL extension (plugin) that converts Postgres into a full-fledged time-series database. TimescaleDB added automatic table partitioning (called "hypertables"), time-series specific query optimizations, and custom functions for gap-filling, downsampling, etc., all on top of vanilla Postgres. This gave users the best of both worlds: the reliability and SQL richness of Postgres, with the performance and convenience of a purpose-built TSDB. The approach proved successful; Timescale (the company) often cites that many users prefer not having to run a separate InfluxDB or Prometheus datastore when they can manage time-series inside Postgres. In fact, as one comparison by Timescale's team showed, **Postgres+Timescale can handle high-cardinality time-series inserts better than InfluxDB in some cases** – for

very large numbers of distinct series, Influx's performance degrades due to its storage engine choices, whereas Timescale (leveraging Postgres B-tree indexing) scaled more smoothly . This suggests that the specialized engine's advantage isn't absolute; with careful engineering, a general engine can catch up or even excel in certain regimes.

Recognizing this, other mainstream databases also moved to incorporate time-series features. MongoDB introduced time-series collections (with automatic bucketing and retention policies) in its core server, eliminating the need for users to hack it for time-indexed data. Microsoft's Azure Data Explorer (Kusto) is a time-series/analytics DB that is now being integrated more tightly with SQL Server for a unified experience. Even Redis (often used for caching) added a TimeSeries module to store time-stamped values efficiently. **The net effect is that by 2025, storing and querying time-series data is considered a standard feature, not a domain requiring a separate product.** PostgreSQL with Timescale extension is a prime example of a general-purpose DB absorbing a TSDB role: it has been adopted in many industrial IoT and monitoring solutions instead of standalone TSDBs. The Rapydo futurist blog imagines by 2035, PostgreSQL might *"natively incorporate … time-series storage and querying as first-class features"*, with the core engine treating these as just another built-in capability . Already, from a user's perspective, the integration is seamless – one can run SQL queries with time-series functions on a Postgres database and achieve what previously required a dedicated TSDB. This trend suggests that **specialist time-series databases could fade** except for extremely high-scale use cases. Indeed, InfluxData (the company behind InfluxDB) has repositioned parts of its stack (IOx engine) to be more general or to integrate with analytics, possibly in response to this competitive pressure.

Vector Databases absorption: The vector database trend provides a dramatic illustration of how quickly a "hot" new specialized category can be folded into existing platforms. In 2023, vector DBs were heralded as the next big thing – purpose-built indexes (like HNSW, IVF) for similarity search on embeddings, which are critical for AI tasks like semantic search or GPT-augmented retrieval. Startups like Pinecone, Weaviate, Zilliz (Milvus) raised large funding rounds, and early adopters rushed to try them out . However, even as investment poured in, a parallel movement started: adding vector search capabilities to established databases. By late 2023 and into 2024, we saw **Postgres get pgvector**, a popular extension to store vector data in a column and perform similarity (nearest neighbor) searches . We saw **MongoDB add vector search** (as a feature in Atlas, its cloud service) . We saw **Redis integrate vector indexing** as a core feature (given Redis's in-memory design, it can do fast vector ops) . Even OpenSearch (the Elasticsearch fork) introduced a KNN vector search plugin. These moves drastically undercut the rationale for a separate vector database in many cases: if a team already has data in Postgres or MongoDB, they can simply enable vector capabilities there, avoiding the complexity of running a new system. The Oso blog *"Feature or Product?: Vector Databases"* captures this well, noting that *"for vector search, pursuing a consolidated strategy appears to be the norm"* by 2024, and asking pointedly if standalone vector DBs are truly an independent category or just a feature . It points out that three of the most adopted databases (Postgres, Mongo, Redis) all jumped on the vector search need quickly, leveraging their massive communities and maturity . In fact, *Retool's State of AI 2024 survey found significantly higher usage of Postgres+pgvector and MongoDB for vector storage than of specialized vector DBs like Pinecone* . The reasons

were clear: using an existing DB is *"cleaner (and familiar)"*, requires no new query language, and benefits from the existing ecosystem and stability of those databases . Moreover – critically – **the performance of these generalist solutions has proved "good enough" for many cases**. Supabase (a cloud Postgres provider) published benchmarks showing pgvector performance scaling well, and Oso's analysis cites that *"solutions like pg_vector are competitive against [Pinecone] on performance"* . In other words, while a specialized vector DB might still win a pure speed contest at the very high end, the gap is not huge for common workloads. The Oso article even notes that the typical advantage of specialized systems (solving a problem that generalized systems can't do or are too slow at) is not strongly present here: *"most databases can feasibly integrate a vector index … and because of how vector search works, on-disk optimizations matter less, with many indexes needing to reside in RAM"* . These are conditions that favor integration into existing DBs (which can use in-memory indexes too).

All major relational databases are now adding or have added vector support: MySQL HeatWave's latest version includes an in-engine vector store with automatic indexing , Oracle announced Oracle DB 23c with native vector data type and indexes, SQL Server is previewing vector functions, and open-source projects like Milvus are pivoting to offer their engine as a plugin to others as well. There is even a collaborative effort to bring standard SQL extensions for vector search, which would further commoditize the capability. The implication is that by the late 2020s, **vector search will be seen as a standard feature of databases, much like text search or JSON support became standard.** Standalone vector databases might remain for users who need absolute top performance or very specialized features (for example, certain vector DBs offer hybrid filtering+vector queries highly optimized, or built-in ML model integrations). But mainstream use – say a developer needing to add a semantic search feature to an app – will likely not require spinning up a separate Pinecone cluster; they can enable pgvector in their existing multi-model DB.

It's worth noting that we have a historical parallel: enterprise search engines and text indexing were once a separate realm (Lucene/Solr, ElasticSearch). Nowadays, even those have partly been subsumed (many databases have text indexing built-in; for moderate needs, one doesn't necessarily deploy Elastic). We may see vector search follow that trajectory even faster. Indeed, Oso's piece concluded that the industry likely **over-hyped standalone vector DBs** and that given the operational cost of new infrastructure, *"the juice isn't worth the squeeze"* for most to adopt a separate vector DB . Unless a vector DB startup can offer something truly unique (perhaps a proprietary embedding generation or a dramatically faster approach), incumbents will continue to erode their differentiation by improving general platforms. Supporting this, *MongoDB's acquisition of an AI vector search startup (Voyage AI) in 2023* suggests Mongo's strategy is to bake the latest vector innovations directly into its general-purpose database , rather than concede that space to niche players.

The broader pattern: feature vs product. The vector DB debate has popularized a decision framework: determine if a new data capability can be implemented reasonably in an existing system with acceptable performance; if yes (and extreme optimizations aren't generally required), then it's a "feature" rather than a new "product" category . Both time-series and basic vector search meet these criteria – they were implementable in existing DBs (via extensions or

new index types), and for most use cases the performance is acceptable after those integrations. Thus, they are becoming features. Only when specialized needs truly can't be met by generalists do we see a separate product being justified (for example, the need for a *columnar* storage led to specialized OLAP databases because row-based OLTP systems were extremely inefficient for big aggregates – that was a genuine case for a new product category) . Even that line is blurring with HTAP databases and systems like DuckDB bridging OLAP into general use.

In conclusion, the evidence strongly indicates that **time-series and vector databases are on a path to being absorbed into general-purpose database engines** by 2030. We will likely talk less about "do I need a time-series database for this project?" and more about "which time-series functions or extensions should I enable on my database?" The standalone TSDBs and vector DBs may either pivot (offering themselves as cloud services on top of engines, or focusing on extreme performance niches) or get acquired and integrated. For users, this consolidation is largely positive – it simplifies architecture and reduces the learning curve. However, it also means that competition shifts to which general database can implement these features most seamlessly and efficiently. Already, Postgres's head start with extensions like Timescale and pgvector has reinforced its popularity as a "one-size-fits-most" solution. As one 2024 blog put it: *"Postgres can indeed work as a capable vector database with the help of key extensions"*, enabling AI search without needing a separate system . The writing on the wall is that bespoke time-series and vector stores will increasingly be seen as *optional* rather than mandatory components in a modern data stack – their functionality lives on, but embedded in multi-model platforms.

Cloud Databases: API Compatibility Trumps Engine Branding

One of the clearest indicators of the "post-bespoke" mindset is how cloud providers package and market their database services. In the early days of cloud (2010s), providers offered well-known database engines in managed form (e.g. AWS RDS for MySQL, PostgreSQL, Oracle, SQL Server). But as cloud vendors developed their own high-performance or globally distributed databases, they faced a challenge: customers were reluctant to adopt something entirely new that wasn't compatible with existing tools and skills. The solution has been to **market cloud databases primarily by their compatibility with popular APIs** – effectively letting the cloud providers innovate under the hood while presenting a familiar face to developers. This strategy has been remarkably consistent across AWS, Azure, and GCP: the emphasis is on being "MySQL-compatible", "PostgreSQL-compatible", "MongoDB-compatible", etc., sometimes more so than highlighting the proprietary engine name.

For example, Amazon's flagship cloud database products are Amazon Aurora and Amazon DynamoDB. Aurora is not a direct port of any existing database – it's a cloud-native storage layer and engine created by AWS – yet **AWS always describes Aurora in terms of MySQL and PostgreSQL**. The service is literally offered in two editions: *Aurora MySQL-Compatible* and

Aurora PostgreSQL-Compatible. AWS documentation proudly states that *"Aurora is fully compatible with MySQL and PostgreSQL, allowing existing applications and tools to run without modification."* . The AWS console even describes Aurora as *"a MySQL- and PostgreSQL-compatible enterprise-class database"* . This framing tells customers: you get the performance and availability enhancements of Aurora, but you don't have to learn anything new or rewrite code – it **speaks the same protocol and language as your existing MySQL/Postgres**. This focus on compatibility has paid off; Aurora became one of AWS's fastest-growing services, precisely because it lowered the barrier for adoption of a "new" engine by disguising it as an old one. As one AWS whitepaper noted, *"Amazon Aurora was designed as a drop-in replacement for MySQL 5.6"* , indicating how central wire-protocol and behavioral compatibility were to its design.

AWS took the same approach for non-relational services. Consider Amazon DocumentDB – AWS did not adopt MongoDB's server (due to licensing issues), but built its own document database service that implements the MongoDB API. They explicitly brand it *"Amazon DocumentDB (with MongoDB compatibility)"* . The service marketing makes it clear that **the key feature is MongoDB API compatibility**, to the point of including it in the product name. Another AWS service, Amazon Keyspaces, is a serverless Cassandra-like database. Rather than push a new query interface, AWS sells it as *"Amazon Keyspaces (for Apache Cassandra)"* , again highlighting that users can use Cassandra drivers and CQL as if it were Apache Cassandra. In effect, AWS's strategy has been to leverage the popularity of open-source DB APIs to quickly gain user trust for their cloud-native reimplementations. They realize that in cloud, *customers care less about the pedigree of the engine and more about seamless integration with their applications*. By offering a buffet of "managed [familiar database]" services, AWS reduced the need for customers to run databases on VMs themselves. It's telling that AWS's database product announcements seldom introduce entirely new query languages; instead, even innovative products like Amazon QLDB (a ledger database) use SQL-92, and Timestream (time-series DB) uses a variant of SQL – again sticking to familiar patterns.

Azure and Google have similar tactics. Microsoft Azure offers **Azure Database for PostgreSQL**, **Azure Database for MySQL**, and **Azure Managed Instance for Apache Cassandra**, all of which are basically managed open-source engines with some Azure tweaks. Even when Microsoft created Cosmos DB – a multi-model, globally distributed database – it provided it with multiple API endpoints: including MongoDB API and Cassandra API, to capture users of those communities. Google Cloud Spanner is a unique globally-distributed SQL database, but after years of pushing its proprietary interface, Google in 2022 added a PostgreSQL compatibility mode to Spanner. This was a major acknowledgement that *speaking Postgres could attract far more developers* since they can use Spanner as if it were a Postgres instance (with some limitations). CockroachDB, a Spanner-like distributed SQL database in the open-source world, likewise invested heavily in **PostgreSQL wire-protocol compatibility**, recognizing that piggybacking on Postgres's ecosystem (ORMs, drivers, skills) dramatically lowers adoption friction .

The trend is so strong that even purely on-prem or open-source projects often advertise their compatibility with a popular DB. For instance, TiDB (PingCAP's database) is MySQL compatible,

and YugabyteDB is PostgreSQL compatible. This “compatibility as a selling point” reflects the commoditization of database interfaces: SQL (and specific dialects like Postgres’s) or Mongo’s JSON API are now seen as standards. Customers want to leverage standard tooling, and avoid vendor lock-in that comes from proprietary APIs. A Percona market analysis in 2025 observed that *“major cloud providers like AWS, Google, and Azure now offer managed PostgreSQL services with proprietary features,”* which was less common five years prior . In other words, the cloud giants realized they needed to embrace PostgreSQL (and other popular OSS databases) rather than fight them. They differentiate by proprietary enhancements and integrations, but *not* by introducing a whole new database language. The same report warns that this increasing number of “proprietary PostgreSQL” variants poses lock-in risk – which ironically further motivates customers to insist on open compatibility in contracts. Many enterprises now mandate in RFPs that any proposed cloud database must be fully compatible with either an open standard (SQL) or a de-facto standard (PostgreSQL/MySQL API), to ensure they aren’t trapped. For example, U.S. government procurement often requires solutions based on open-source or standard interfaces to prevent dependency on a single vendor technology. A real-world illustration: a HashiCorp Terraform guide explicitly allows Amazon Aurora (Postgres edition) to fulfill its requirement for a PostgreSQL database . This implies that in the eyes of enterprise software, **Aurora Postgres = Postgres for all practical purposes**. It’s a validation that what matters is the interface and behavior, not who wrote the code.

Top cloud DBs in 2026–2030 will therefore emphasize API compatibility over engine brand. We foresee marketing materials and documentation continuing this trend. For instance, AWS might introduce new features or services but still frame them as enhancements to “PostgreSQL-compatible Aurora” or “MySQL-compatible Aurora”, rather than touting a wholly new brand. Google could integrate more databases under the Google Cloud SQL umbrella (which supports MySQL, Postgres, SQL Server). IBM Cloud and Oracle Cloud similarly offer databases named after Db2 or MySQL, etc. The advantage for cloud vendors is clear: they tap into existing communities. For users, this means easier migration to cloud (as their apps work with minimal changes). It also means the traditional concept of a “database product” is shifting – the *protocol* is the product, in a sense. We can think of database protocols (Postgres, MySQL, MongoDB, Cassandra, etc.) as standards like SQL or ODBC used to be. Cloud vendors are effectively competing on who can provide the best service *for a given standard*. It’s analogous to how one might choose an email provider: multiple providers implement the same protocols (SMTP, IMAP) but compete on reliability, storage, etc., rather than inventing new email protocols.

An interesting question is whether cloud providers will even try to push their engine brand at all, or simply hide behind compatibility. Oracle’s strategy with MySQL HeatWave is an example of a vendor trying to differentiate while still leveraging MySQL’s brand: Oracle built a new cluster architecture and in-memory analytics for MySQL (HeatWave), but rather than call it something entirely different, they integrated it *into MySQL service* and highlight it as *“the MySQL that does both OLTP and OLAP”*. In their research piece, Oracle boasts MySQL HeatWave’s performance vs other analytic databases , but they make it clear you use it via standard MySQL protocols (just with HeatWave turned on). So even Oracle, which historically had no issue pushing

proprietary tech, recognized the value of compatibility (here, sticking with MySQL's interface to lure MySQL users to Oracle Cloud).

Another angle is multi-cloud and portability: if your cloud DB has a standard interface, you can advertise it as "move your data to our cloud with zero code changes, and move out if you want – we're standard." AWS, for example, might downplay that second part but the first part is heavily advertised (Aurora's ease of migration from MySQL). Google's embrace of PostgreSQL in Spanner was partly to assuage concerns that Spanner was too proprietary – now at least if you move to Spanner's PG mode, you could theoretically move off to another PG system later, protecting you from total lock-in.

API compatibility focus also appears in RFPs and procurement (tying into the next section). We note that when enterprises solicit proposals, they increasingly specify *"support for PostgreSQL"* or *"must integrate with MongoDB API"* rather than naming specific vendor products. This allows them to consider cloud or on-prem solutions interchangeably. For example, a government RFP might say "the solution's database must be PostgreSQL 13 or compatible" – which means a vendor could propose AWS Aurora or Azure Postgres or on-prem Postgres, all fulfilling that requirement. This is a stark change from a decade ago when an RFP might have said "must run on Oracle 12c" (tying to a product). It shows how compatibility has become a *de facto* requirement in many cases.

In conclusion, by 2026–2030, **the top cloud database services will be largely defined by the standard APIs they expose**. The actual engine brand (Aurora, Spanner, Cosmos, etc.) will be secondary in messaging. Cloud providers will continue to innovate under the hood – developing custom storage engines, separation of compute and storage, serverless autoscaling, global distribution, etc. – but all that will be presented behind familiar interfaces like PostgreSQL or MongoDB. The "database era" thus shifts from one where customers picked a *product name* to one where they pick a *service describing an interface*. A technical decision-maker in 2028 might say, "We chose a PostgreSQL-compatible cloud database that meets our scalability needs," without necessarily being concerned whether it's Aurora or Cloud SQL or a self-driving Postgres variant, as long as it's PostgreSQL-compliant. This marks a significant fade of bespoke-ness: the emphasis is not on creating novel query languages or specialized APIs, but on *delivering improvements under the cover of standard ones*. The brand value of proprietary databases diminishes unless tied to compatibility (e.g., "Aurora" mainly has value because it is associated with MySQL/Postgres compatibility plus better performance).

One can even envision cloud providers offering *"API-level SLAs"* – for instance, AWS could position that it offers the best "MySQL API" service (Aurora) and the best "Mongo API" service (DocumentDB or a future replacement), and so forth. This is already implicitly happening. The ultimate outcome is that the **engine brands fade into the background**. Users talk about "Postgres on AWS" or "Mongo on Azure" even if under the hood it's not the community edition software. This trend strongly reinforces the decline of niche databases: if a new database can't align to an existing API, its adoption barrier is very high in the cloud era. Conversely, if a new database can pretend to be an old one (like ScyllaDB pretending to be Cassandra), it has a foothold.

Protocol-Compatible Alternatives and the Fate of Forks

One measure of how the database landscape is consolidating around common interfaces is the fate of database **forks** and the rise of **protocol-compatible alternatives**. In the open-source world, “forks” occur when developers split off from an existing project to create their own variant (usually retaining compatibility at least initially). MariaDB’s fork from MySQL in 2009 is a prime example – it was created to ensure a fully open-source path when Oracle acquired MySQL. MariaDB started as a nearly drop-in replacement for MySQL, and over time diverged somewhat with its own optimizations and features. Percona Server is another fork (of MySQL) aimed at performance and diagnostics. These forks saw some adoption (MariaDB even replaced MySQL by default in certain Linux distributions, and has its own user base). However, the trend emerging in the 2020s is that even forks can be overtaken or *“replaced”* by totally separate implementations that maintain the **same protocol or API**. In other words, instead of continuing the lineage of the original code via forks, the market often jumps to a new solution that is protocol-compatible.

The question **“How often do forks get replaced by protocol-compatible alternatives?”** can be interpreted as: when an ecosystem has a popular DB (DB X), and one or more forks (X1, X2) exist, how frequently does an outsider (DB Y) come in that isn’t a fork but implements X’s interface, and ends up supplanting the original or its forks? The answer appears to be: with increasing frequency in recent years, especially as big players leverage this strategy. For example:

- **MySQL Ecosystem:** MySQL has two major forks: MariaDB and Percona. MariaDB in particular positioned itself as a “community-driven drop-in replacement” for MySQL, and achieved some success (especially 2012–2017, where it was adopted by Wikipedia and others). However, Amazon’s Aurora for MySQL (not a fork in the open-source sense, as its code isn’t public, but a rewrite at least of the storage layer) arguably *leapfrogged* the traditional MySQL forks in adoption on the cloud. AWS customers that might have considered migrating to MariaDB for performance or features instead often choose Aurora because it offers greater performance improvements (5× MySQL throughput) and a managed service convenience . Aurora is presented as a “MySQL-compatible” target for migration , making it an easy switch. So in effect, a *closed-source reimplementation* by AWS has outpaced the community forks in growth. By 2022, MySQL’s total user base (especially on cloud) includes a large portion on Aurora MySQL, which **is not MySQL code but is protocol-compatible**. TheCUBE research highlighted that MySQL’s market share figures (20.6% of transactional DB market) don’t even count MariaDB, Percona, or cloud implementations – meaning the combined MySQL-compatible share is even higher . Many of those users may not even realize they’re not running Oracle’s MySQL code anymore (since from their perspective it behaves the same). So this is a prime example: the original MySQL and its forks (Maria, Percona) have effectively been *augmented or overshadowed* by Aurora and other MySQL-compatible services, an alternative platform that replaced the need for maintaining a fork for many customers.

- **Cassandra Ecosystem:** Apache Cassandra did not have prominent forks (it's governed by Apache, though DataStax had a proprietary version). But an alternative named **ScyllaDB** emerged (circa 2015) which is *not* a fork of Cassandra's Java code, but a rewrite in C++ designed for much better performance and lower latency. ScyllaDB kept the Cassandra SSTable format and CQL query language for compatibility . Essentially, it positioned itself as "Cassandra, but 10× faster and written in C++ with an actor-model architecture (Seastar framework)" . Over the following years, Scylla gained users who required high throughput (it boasts millions of ops/sec on a single node, something Cassandra struggled with) . We can see this as an alternative engine *replacing* Cassandra deployments while keeping the Cassandra interface. Companies that might have considered forking or heavily modifying Cassandra instead just migrate to Scylla for performance, since their application doesn't need to change (CQL is the same). Although Cassandra remains widely used, Scylla's existence as a drop-in alternative shows how a protocol-compatible solution can divert the user base.

- **MongoDB Ecosystem:** MongoDB's case is interesting due to licensing. When MongoDB, Inc. changed its license to SSPL in 2018 (which is not OSI-approved open source), it prevented cloud providers from offering MongoDB-as-a-service without either licensing from Mongo or using an older version. In response, AWS did not fork the SSPL code (which they legally couldn't do beyond older versions); instead, they created **DocumentDB** which mimics MongoDB's API. DocumentDB started compatible with Mongo 3.6 API and has progressed from there. It is not a fork of Mongo's code; it's internally a different engine (based on PostgreSQL for query processing, some sources suggest, and a distributed storage underneath). Amazon thus provided a protocol-compatible alternative to MongoDB that many AWS customers use if they want a managed Mongo-like service without using Mongo's own Atlas service. Additionally, an open-source project **FerretDB** emerged to provide a MongoDB-compatible layer on Postgres for those who want to avoid Mongo's license entirely in self-hosted scenarios. Microsoft went a step further – as noted, in 2023 it launched an open source project codenamed **"DocumentDB"** (unfortunate naming collision with Amazon's service) which is a **MongoDB-compatible database built on PostgreSQL** in partnership with FerretDB . This suggests that even for MongoDB (which isn't easily forkable now), multiple independent implementations of its protocol have appeared. If these gain traction, they effectively *replace the original in some deployments*. Indeed, the Percona report hints that many are moving away from MongoDB's own platform due to dissatisfaction and cost, with some choosing PostgreSQL as an alternative for similar needs . A blog cited in that report called "The Great Migration from MongoDB to PostgreSQL" described how a startup dumped Mongo for Postgres due to Mongo's limitations and cost, and saw 50% cost reduction . In that case, they didn't use a Mongo-compatible new engine, they just ported data to Postgres JSONB (so protocol compatibility wasn't maintained, they actually switched APIs). That is another route – sometimes the alternative is not protocol-compatible but functionally overlaps enough (Postgres JSONB can replace some Mongo use cases). Either way, Mongo's story indicates that *if a proprietary or specialized DB stumbles (through license or tech), the*

community finds alternatives, whether through forks or through new implementations or through different solutions entirely. Over time, those alternatives can supplant the original. We may see by 2030 that a sizable chunk of "MongoDB API usage" is actually on non-Mongo engines (AWS, Azure, FerretDB, etc.), just as a chunk of MySQL usage is on Aurora or MariaDB.

- **Oracle and SQL Server:** Historically closed databases like Oracle have had some compatible imitators (e.g. EnterpriseDB's Postgres Plus Advanced Server offered Oracle PL/SQL compatibility to lure Oracle users to Postgres). These haven't fully displaced Oracle, but they show attempts at protocol/API replacement. As open-source and cloud databases keep improving, even Oracle workloads sometimes migrate to PostgreSQL (with Oracle compatibility tools), which is essentially replacing Oracle with an alternative that tries to behave like Oracle enough to run the app.

The increased **frequency of these protocol-level swaps** is notable. In the past, migrations between databases meant significant application rewrites (e.g. moving from Oracle to MySQL required changing SQL dialect, features, etc.). Now, with the prevalence of look-alike systems, companies can migrate with far less pain: MySQL -> MariaDB or Aurora (no query changes), Cassandra -> Scylla (no query changes), MongoDB -> DocumentDB (drivers the same). This lowers switching costs, which in turn means incumbents must compete on merit rather than relying on technical lock-in. It also means if a fork exists but doesn't keep up, a whole new impl can bypass it. For instance, MariaDB faces competition not just from MySQL (Oracle's version) but from Aurora – and the latter's resources (AWS R&D) dwarf what MariaDB Corp has, so we see AWS out-innovating what a fork could do (Aurora's architecture yields performance MariaDB alone couldn't achieve on the same scale). So yes, forks (which typically share the original's limitations up to a point) can be leapfrogged by novel architectures that still maintain compatibility.

To quantify "how often": we can cite MySQL's numbers as a case. MySQL's usage including all variants is huge; the fact that Aurora and others aren't counted in MySQL's official market share means a big chunk has been carved out by alternative engines . Similarly, a lot of Cassandra's largest deployments (at Apple, etc.) have started evaluating Scylla to replace some clusters. These replacements happen especially when performance or cost is a concern. Another example: *Kafka vs Redpanda.* Apache Kafka (not exactly a DB, but a streaming data system) has no forks of note, but Redpanda came as a drop-in Kafka API-compatible event streaming platform (written in C++ without ZooKeeper). Redpanda has gained adoption by promising lower latency and simpler ops, while applications use the Kafka API as before. This mirrors the DB trends, showing the model of "new implementation, same API" is popular across data infrastructure.

From a governance perspective, these protocol-compatible alternatives often arise when the original project is either controlled by a single vendor (and thus slow or has a licensing change) or when technological advances allow a fresh start to outperform legacy code. AWS's motivation was partly licensing (Mongo, Elastic) and partly performance (MySQL, Cassandra). The

community's motivation for FerretDB was licensing. Scylla's motivation was performance by using modern C++ and per-core architecture. Each time, the API-level consistency is a deliberate choice to capture users. This happens **increasingly often because the interfaces of databases have standardized** (SQL, or specific popular dialects).

In a world where, say, every application used completely different database APIs, such swaps would be rare. But now, with a handful of common interfaces, alternative products can cluster around those interfaces. We have a *"cluster" of products around the PostgreSQL interface* (Postgres itself, Amazon Aurora Postgres, Cockroach, Yugabyte, etc.), a cluster around MySQL interface (MySQL, MariaDB, Percona, Aurora MySQL, SingleStore to some extent, etc.), one around MongoDB API (MongoDB, DocumentDB, Azure's API, FerretDB), one around Cassandra API (Cassandra, Scylla, Cosmos DB Cassandra API, Keyspaces). Within each cluster, switching is easier. So forks may have been an earlier phenomenon (staying within same code family), but the new phenomenon is cross-implementation switching within an interface family. It's happening quite often as outlined.

Thus, **the share of deployments that use a "protocol-compatible alternative" instead of the original fork is rising.** As of 2022, for MySQL protocol, alternatives like MariaDB, Aurora, etc., together probably account for a significant minority of instances. TheCUBE noted explicitly that the count of MySQL installations doesn't include those forks and cloud versions , implying the true footprint of the MySQL protocol is larger and spread across multiple engines. We expect this pattern to replicate for other ecosystems. By 2030, one could imagine that for any given major database API, there are multiple interchangeable implementations in play, and it's not unusual for the original vendor's implementation to *not* be the one in use. This is basically the commoditization of database protocols.

One real-world sign of this: a company might migrate an OLTP workload from on-prem Oracle to AWS Aurora (Postgres mode) to save cost – effectively replacing Oracle's engine with Postgres-compatible one. Or migrate from self-managed MongoDB to Azure Cosmos DB's Mongo API for scalability – replacing the back-end but keeping the API. These migrations are increasingly mentioned in case studies and community discussions, suggesting it's not a rare thing anymore.

To summarize, **protocol-compatible replacements are now a common outcome in the life cycle of database technologies**. Many forks that started as independent (like MariaDB) now find themselves one of several compatible options, rather than the only alternative. The frequency of such replacements is tied to the maturity of the protocol: the more stable and widely-understood the API (e.g. Postgres, MySQL, Mongo), the more likely multiple competing engines will implement it, giving users choices. As databases continue to standardize around a few interfaces (SQL variants or popular NoSQL APIs), this dynamic will only increase. New innovations will often be delivered as *alternative engines under old APIs*, not as entirely new APIs – because that's the fast track to adoption.

From a decision-maker's viewpoint in 2026–2030, this means that choosing a database technology is somewhat decoupled from choosing a specific vendor's product. You might decide

“we’re a PostgreSQL shop” – but whether the runtime is PostgreSQL OSS, Aurora, Cloud SQL, or some future distributed Postgres-compatible system can be decided based on other factors (cost, performance, support) and can change over time without rewriting the application. This is a powerful position for users to be in, and it further weakens the argument to go with a bespoke specialized DB that doesn’t have multiple implementations or broad compatibility – because that *would* lock you in. We can see how this ethos reinforces using common APIs and therefore favoring the multi-model generals that support them.

Performance Gap: Generalist vs. Specialist Databases

One of the historical arguments for bespoke databases has been performance: a database optimized for a specific type of workload should outperform a general-purpose system that tries to do many things. A decade ago, this was often true by a large margin. For example, a specialized time-series DB could ingest data far faster than a traditional RDBMS with row-oriented storage, or a graph database could traverse relationships much quicker than a relational join-based approach. However, **the performance gap between generalist and specialist databases has been steadily narrowing** on many key workloads. By leveraging new technologies (in-memory processing, columnar formats, vectorized execution, etc.) and through sheer engineering effort, general-purpose databases have significantly improved their performance in areas that used to be weaknesses. Additionally, hardware advancements (SSD latency, plentiful RAM, many-core CPUs) have raised the performance “floor” so that even less optimized approaches can often be *fast enough* for many use cases. In short, the premium one paid in complexity for that last bit of performance is no longer always worth it, unless you truly need maximum throughput.

Let’s examine a few workloads and how the gap has changed:

- **Analytical Queries (OLAP) vs OLTP Databases:** Traditionally, analytical databases (column-stores like Teradata, Vertica, or more recently Snowflake, ClickHouse) were 10–100× faster than row-based OLTP databases at large scans and aggregates. General-purpose relational DBs were not competitive for serious data warehousing workloads. But the emergence of **HTAP (Hybrid Transactional/Analytical Processing)** is addressing that gap. A prominent example is **Oracle MySQL HeatWave**, which Oracle claims *“converges OLTP and analytics into one unified MySQL database”*, accelerating queries to the point MySQL can do warehousing tasks without exporting to a separate DB . Oracle’s published benchmarks (to be taken with a grain of salt) showed HeatWave MySQL beating Amazon Redshift, Snowflake, etc., on certain analytics, while still being MySQL for transactions . If MySQL – a general OLTP database – can perform in the ballpark of specialized analytic databases for many queries, it erodes the necessity of having a separate analytics DB for moderate workloads. Similarly, SQL Server and PostgreSQL introduced columnar indexing (SQL Server’s ColumnStore index, Postgres’ Cstore extension or Greenplum approach), to speed up analytics on primary data. MariaDB added ColumnStore, and Amazon Aurora is experimenting with distributed parallel query processing. These enhancements mean that the gap between an

operational DB and a specialized warehouse is smaller, at least up to certain data sizes. For real-time analytics on recent data (say last month's transactions), one might not need a separate data warehouse if their main DB can run fast analytic queries. Oracle's integration of HeatWave into MySQL exemplifies this narrowing: *complex queries that were "atrocious" in vanilla MySQL become feasible in MySQL HeatWave* , eliminating ETL delay and duplication .

- **Full-text Search:** This is another domain where specialized engines (Elasticsearch, Solr) used to be the only viable option for speed. Now, many general databases have built-in full-text indexes (Postgres, MySQL, SQL Server all do). While Elastic might still be more feature-rich for search, the gap in simple search query performance isn't as huge for moderate scales. Thus some apps drop the separate search engine and use Postgres text search to simplify architecture, accepting maybe a bit less power.

- **Time-Series writes:** As discussed, InfluxDB vs TimescaleDB is illustrative. InfluxDB's engine is highly optimized for appending time-series data. Timescale (Postgres) initially lagged behind for insert throughput in low-cardinality scenarios (few series, high freq), but for higher cardinalities (many series), Influx's write performance *"drops dramatically"* while Timescale's approach handles it better . This means at scale, Timescale on Postgres can rival or beat Influx for ingest, narrowing that gap. Also, improvements like partitioning, copy pipelines, etc., in general RDBMS have improved bulk insert rates.

- **Vector similarity search:** We touched on performance above. Specialized vector DBs tout optimized indexes (HNSW, IVF) and scale-out for very large embedding corpuses. But Postgres with pgvector can also use HNSW and can leverage high-memory instances. Supabase's benchmarks and others show pgvector performing surprisingly well. The Oso analysis even said *"pg_vector is competitive on performance"* vs Pinecone . Weaviate (a vector DB) founder admitted in a blog that for smaller datasets, pgvector is fine; the big difference might appear at larger scale or if needing horizontal sharding. Thus, for many AI applications, adding a pgvector index yields sufficiently fast query times (millisecond-range) that a separate vector DB may not be justified. We also saw that one specialized vector DB, Chroma, achieved high speed mainly by being in-memory (essentially acting like a cache) . But any general DB can also be run mostly in-memory if you throw RAM at it, which again narrows the advantage unless memory is limited.

- **Geospatial**: Specialized GIS databases (like ESRI or spatial extensions) were once required for performant geospatial queries. Now, Postgres with PostGIS extension is extremely capable and often used instead of separate spatial databases. Its performance is sufficient for many and it's integrated with SQL queries.

- **Hardware Utilization:** Historically, specialized DBs often were built to exploit hardware trends (e.g., Scylla using many cores and async I/O better than Cassandra, or in-memory DBs using RAM). General DBs have caught up by incorporating similar

techniques. For example, modern Postgres can use parallel query execution across CPUs, something not present years ago. MySQL has Group Replication and buffer pool improvements. Additionally, the widespread availability of SSDs and NVMe has improved the I/O performance of general-purpose DBs significantly (removing one advantage specialized engines had when they assumed spinning disks needed special append-only designs, etc.). The AWS Prescriptive Guide mentions how Exadata (Oracle's specialized hardware) improved throughput, but at high cost . Now, cloud databases on commodity SSDs can match a lot of that performance without exotic hardware . The cloud, by offering easily scalable instance types (lots of CPU or RAM), also allows boosting a general DB's performance by brute force scaling vertically or horizontally in ways a single on-prem instance couldn't. Aurora, for instance, decouples storage and uses a distributed storage service to improve performance and recovery – these architectural changes close gaps with bespoke high-end systems (Aurora can approach commercial DB performance at lower cost).

To concretize with an example: Suppose we have a workload of storing and querying 100 million time-series metrics with both high ingest and queries for recent trends. In 2015, one might *need* something like InfluxDB or Cassandra to handle that, as MySQL or Postgres would choke. In 2025, one could use TimescaleDB on a cloud Postgres cluster or AWS Timestream (which is specialized under the hood but offered as a service with SQL) – either way, the mainstream solutions can handle it, perhaps needing a beefy instance, but it's feasible. The difference might be that a tuned Influx on the same hardware might still be faster, but if Postgres is "fast enough" and you avoid complexity, you might choose Postgres.

However, it's important to note that **the gap narrowing is not universal**. For truly massive scale or extreme low-latency needs, specialized systems still have an edge. For example, ultra-high-frequency tick data (millions of writes per second) might still require a purpose-built solution or a specialized distributed log like Kafka + ksqlDB rather than a general DB. The Oso blog's insight on vector DBs acknowledges that for something like Chroma (in-memory vector DB), it delivered 40–80ms query latencies where Postgres might be slower if not fully memory-resident . So specialists can still outperform. But the key shift is that the *performance difference is often no longer a decisive factor for moderate workloads.* As one metric, if a general DB can do a task in say 50ms and a specialized one can do it in 5ms, but the application requirement is 100ms, then either is fine. The user may prefer the general DB for simplicity. The **"good enough" threshold** is now reachable by general DBs for many tasks.

Vendors of general DBs have actively targeted narrowing these gaps because they know eliminating the need for a second system is highly attractive to customers. For instance, Microsoft SQL Server added **HTAP features** (memory-optimized tables, clustered columnstore indexes) to claim it can do both OLTP and OLAP on the same system. IBM Db2 did similar with BLU Acceleration. MySQL HeatWave's marketing is directly *"no need for a separate analytics DB, do it in MySQL"*. These initiatives are testament to generalists chasing specialists on performance turf. And they often cite numbers: Oracle claimed MySQL HeatWave is X times faster than Aurora + Redshift workflow for certain queries , trying to prove that a single system

can beat a pipeline of specialized ones. Even if those numbers are optimistic, it demonstrates the focus on closing the performance gap.

Another area: **New data types and associated performance**. When JSON first came into Postgres, it was slow compared to MongoDB for heavy JSON workloads. Over time, Postgres added JSONB (binary JSON with indexing) and improved it. Now, Postgres can in many cases match MongoDB's query performance for moderately sized JSON datasets (especially if well-indexed) . So a user might not see a need for Mongo if Postgres can store JSON and query it decently – the gap has closed enough. Similarly, for graph queries: pure graph databases (Neo4j, etc.) are optimized for deep traversals. But if your graph queries are relatively simple (2-3 hops), a relational DB with the right indexes or a multi-model like Cosmos DB's Gremlin API might suffice. We've also seen approaches to integrate graph via extensions (e.g., AGE extension brings openCypher to Postgres). If those become faster, the need for a separate graph store diminishes for many use cases.

Hardware trends favor generalists: Cloud infrastructure means one can scale vertically easily (choose a bigger instance with more RAM/CPU). Many performance problems can be alleviated by scaling up hardware, which is often easier than redesigning architecture with a specialized DB. With RAM prices dropping, you can often just put your entire working set in memory on a single node (for a lot of workloads) and a general DB will perform extremely well. Thus specialized in-memory DBs (like VoltDB, etc.) didn't become mainstream because adding RAM to Postgres or MySQL and using memory-optimized settings gave enough performance in many cases.

That said, for truly **extreme scale (global scale, petabytes)**, specialized or niche architectures still lead (e.g., Snowflake for multi-petabyte analytics, or specialized time-series like Druid/ClickHouse for ultra-fast aggregated analytics). But those are often for analytics rather than operational DBs. In operational sphere, distributed NewSQL databases like CockroachDB show that you can have an almost drop-in Postgres-compatible DB that scales horizontally – which was once the domain only of NoSQL for write scaling. It's not as battle-tested at giant scale as Cassandra, but it's catching up for many uses. The gap in scalability is thus also narrowing.

In summary, by 2026–2030 we anticipate **fewer scenarios where a general-purpose database is completely unsuitable performance-wise compared to a specialist**. Instead, the generalist might achieve, say, 70-90% of the specialist's performance. If that difference is not critical to the business, many will choose the generalist and simplify their stack. The confidence in generalists is growing: as one Rapydo article put it, *"the relational model will continue to prove its versatility"*, and with each generation of improvements, it can handle more and more workloads that used to require alternatives . Indeed, the **market is betting on generalists**: SingleStore, Oracle's converged DB, Azure's Cosmos (which tries to be multi-model & multi-API) – all these exist because there's a belief that one engine can be performant enough across tasks.

It's important to highlight that **counterexamples exist** (we will in the counter-evidence), but the general trend in performance is convergence.

To provide a concrete data point: Oracle's benchmarking claims for MySQL HeatWave showed it being *23-85% faster than Snowflake* on certain queries and *3X cheaper*, while also being *2-3X faster than Amazon Redshift* . If even half true, that implies a general OLTP database (MySQL) augmented with an analytics module can play in the same performance league as specialized analytic platforms. MySQL HeatWave also introduced an in-database **machine learning** feature to train models without exporting data, again aiming at a one-stop-shop with competitive performance to ML platforms. This is a glimpse into the future where the gap is so narrow that using one system is clearly advantageous unless extremely high performance is needed.

Another data point from Oso's vector analysis: they mention *"pg_vector are competitive … on performance"* and also that *"solutions like Pinecone have been riddled with errors"* , implying that sometimes specialized new systems have stability issues that offset their performance edge. If a general system is slower but rock-solid, that also effectively narrows the gap in real-world viability.

In conclusion, **general-purpose multi-model databases are increasingly "close enough" in performance on key workloads that the trade-off of introducing a specialized system purely for performance is often not justified.** The continuous improvements in general DB engines, combined with better hardware, have closed much of the historical gap. While specialists will continue to push the envelope (and generalists will follow), the space where specialists enjoy an indisputable performance mandate is shrinking. For new projects in 2030, a team might default to a multi-model DB and only consider a separate specialized store if profiling clearly shows a severe bottleneck that can't be resolved by scaling the general system. That's a reversal from 2010 when one might default to polyglot – using Mongo for JSON, Hadoop for big analytics, etc., because the general options couldn't cope. Now they often can.

Changing Procurement Criteria: RFPs Specify Interfaces and Patterns, Not Products

As the database market matures and standardizes, the way that enterprises evaluate and request database solutions is evolving. In the past, it was common to see **RFPs (Request for Proposals)** or procurement documents that explicitly named specific database products – for example, "proposed solution must use Oracle Database 12c" or "must be built on Microsoft SQL Server". Those days are waning. Organizations, especially large enterprises and government agencies, have learned the hard way about vendor lock-in and are shifting toward describing requirements in terms of capabilities, interfaces, or patterns rather than specific brands. This allows more flexibility in choosing or swapping the underlying engine, and ensures longevity of the solution as technology changes. By specifying an interface (like SQL or an API) instead of a product, the buyer can obtain multiple competitive bids and keep the option to change implementations later as long as they adhere to that interface.

Several trends drive this change:

- **Avoiding Vendor Lock-in:** If an RFP names a particular proprietary database, the organization is effectively committing to that vendor, potentially for decades. Instead, by requiring a standard interface (e.g. ANSI SQL or PostgreSQL compatibility), the organization can later choose among many vendors or cloud providers that support that interface. This is particularly important in government and public sector, where dependence on a single vendor can be seen as risky or against procurement rules.

- **Rise of Open Standards and Open Source:** With open-source databases like PostgreSQL and MySQL proving themselves in mission-critical uses, many organizations prefer them or at least want the *option* to use them. An RFP that says "must run on an open source relational database or compatible service" is increasingly common. For example, the IETF (Internet Engineering Task Force) in an RFP explicitly noted it was migrating from MariaDB to PostgreSQL , implying future solutions should support or use PostgreSQL. We also see some governments mandate that solutions use open technology unless there's a compelling reason otherwise.

- **Commoditization of Database Technology:** Databases are no longer exotic niche products; a few interfaces cover most use cases. Thus specifying at the interface level is feasible. For instance, instead of saying "we need a graph database such as Neo4j", an RFP might say "the solution should support graph traversal and query patterns (openCypher or Gremlin interface)". This way, a vendor could propose a Neo4j-based solution or a Cosmos DB (with Gremlin API) or even a JanusGraph on Cassandra – as long as it meets the interface/pattern, it's acceptable. The benefit is competition and future-proofing.

- **Cloud and DBaaS offerings:** As discussed, cloud providers often have multiple engine options behind the scenes. When a company issues an RFP, they might not want to exclude a cloud-based answer. For example, if an RFP requires "Oracle database", that essentially locks into Oracle Cloud or on-prem Oracle, excluding AWS or Azure solutions. Instead, if they specify "SQL database with XYZ features, must provide Oracle compatibility or migration path", that opens the field to proposals including non-Oracle tech that mimics Oracle (like AWS Aurora with Babelfish for SQL Server, or EDB Postgres Advanced Server which has Oracle syntax support). Indeed, *HashiCorp's documentation stating a requirement of "PostgreSQL or a PostgreSQL-compatible database such as Aurora"* is a microcosm of how technical requirements are now phrased. They don't insist on the community PostgreSQL specifically; an equivalent service that speaks the same dialect is fine.

We have concrete examples of this shift:

- The HashiCorp Terraform Enterprise doc we saw: *"a PostgreSQL server such as Amazon RDS for PostgreSQL or a PostgreSQL-compatible server such as Amazon*

Aurora PostgreSQL must be used." . This clearly shows that being **PostgreSQL-compatible** is considered sufficient to meet the requirement. In an RFP context, that means if a vendor proposes Aurora instead of vanilla Postgres, it's acceptable because the interface is the same.

- In government solicitations, one can find language like "The solution's database must support ANSI SQL and ACID transactions. Preference will be given to solutions using open-standard database engines (e.g. PostgreSQL or MySQL) or fully managed cloud databases with equivalent compatibility." This sort of wording has started replacing "must run on IBM Db2" etc.

- Some organizations even mandate *data portability.* For example, the EU has guidelines to avoid cloud vendor lock-in which might encourage specifying requirements like "the database must allow export in standard formats and use standard interfaces" – which again steers toward widely compatible databases.

Another factor is **requests focusing on patterns**. For example, rather than saying "we need a Hadoop-based data lake and a NoSQL DB for user profiles and a relational DB for transactions", an enterprise might write an RFP that describes the patterns: "must handle high-volume append-only event storage for analytics (data lake pattern), must handle user profile store with flexible schema (document/NoSQL pattern), and a transactional store for billing (relational pattern)." Then they leave it to bidders to propose an architecture that could involve multiple engines or perhaps a multi-model single engine that covers all patterns. Increasingly, bidders might propose multi-model solutions: e.g., "We will use Azure Cosmos DB for both the profile (document) and event store (using its API) and Azure SQL DB for billing", or another might say "We'll use a single Couchbase or single YugabyteDB for all, as it can handle JSON and transactions." The RFP focusing on patterns rather than product names encourages innovative solutions and doesn't straightjacket the architecture to a specific vendor from the start.

Within organizations, architects writing internal requirements also often speak in terms of capabilities: e.g., "we need a database that can do full-text search, JSON querying, and time-series aggregations." Those capabilities could all be met by a single multi-model DB (like say Elastic + relational combined, or just Postgres with appropriate extensions), or by separate DBs. But they're not immediately saying "we need Elastic and Mongo and Postgres." They articulate the needs and leave room to decide how many engines and which ones.

This shift is partly a response to how quickly database tech changes too. If you write a static RFP around a specific product, by the time the project is delivered that product might be outdated or not best-of-breed. By sticking to API or pattern requirements, you allow adapting to the best implementation available at the time. For example, an RFP in 2025 might say "Document database capability (MongoDB API or equivalent)". This means if, say, in 2027 a better MongoDB-compatible service exists, the implementer can use that, not necessarily MongoDB itself.

We can also see evidence of this thinking in multi-cloud strategies: a company may specify “the solution must be database-agnostic or use open interfaces to allow deployment on different cloud providers.” That implies they want, for instance, Postgres or MySQL compatibility so that they can run on AWS or Azure or on-prem interchangeably. With containers/Kubernetes, some RFPs might even ask for the solution to be delivered with a choice of database backends.

Impact on vendors: Database vendors now often include in proposals how they meet open standards. A proprietary database vendor might highlight that they support standard SQL or compatibility layers (like Oracle can run PL/SQL but also maybe standard SQL). Open-source vendors highlight the lack of lock-in. All of this is a result of customers explicitly asking for that in procurements.

In the context of “Bespoke DB era fades,” this procurement trend is both cause and effect. It’s a cause because by focusing on interfaces/patterns, it favors solutions built on widely-used, multi-purpose technologies (since those are more likely to meet a standard interface). It’s an effect because as multi-model general DBs prove capable, buyers feel they don’t need to call out niche DBs – they assume a standard solution can do it.

Anecdotally, some large enterprise RFPs now forbid naming specific vendor products in requirements (to ensure fairness and avoid inside dealing). They might say “relational database management system (RDBMS) with capabilities X, Y, Z” rather than “Oracle or equivalent.” If they do say “or equivalent,” that’s key. For instance, *“Proposals can use Oracle Database or an equivalent RDBMS that supports PL/SQL”* – this invites a PostgreSQL-based solution with PL/pgSQL maybe. Or *“IBM IMS or equivalent hierarchical database”* in some legacy context, etc. We saw an example: *Terraform requiring Postgres or a Postgres-compatible service* – the phrasing “or compatible” is now common.

RFPs specifying patterns might include things like:

- “The solution should provide a key-value storage mechanism for caching and fast lookup (e.g. Redis or similar) for session management.” Here, they name Redis as an example but leave it open.

- “All data storage components must have a documented API (such as SQL, REST, GraphQL, etc.) and should not rely on proprietary interfaces that cannot be replaced by an alternative implementation.”

- “Preference will be given to solutions employing open-source databases with a strong community (PostgreSQL, etc.), unless a clear case is made for a proprietary alternative due to specific requirements.”

In government RFPs, one might see references to compliance and standards. For databases, SQL (ANSI SQL) is a standard they might reference. Or requiring that the solution adheres to data portability standards, again implying use of common formats and query languages.

The move to cloud also means some RFPs are written as “we want a managed service that does X,” not caring which service specifically. For example, “the application should use a managed relational database service (e.g. AWS RDS, Azure SQL, etc.) supporting PostgreSQL or MySQL, to minimize operational burden.” This doesn’t pin down which cloud or product, just the type and interface.

All of this underscores the fading prominence of individual database brand names in favor of *capabilities and compatibility.* In the bespoke era, an RFP might have been highly specific: “We will use Neo4j for graph, Hadoop for big data, Cassandra for time-series, etc.” Now, it might be: “We need graph querying capability, large-scale data analytics capability, time-series handling – show us how your solution provides these.” The onus is on integrators to choose whether that’s one multi-model DB or multiple; but often they may lean to fewer multi-model solutions to simplify meeting all requirements.

Finally, consider **multi-cloud RFPs**. Some organizations explicitly require that a solution be portable across cloud providers (to avoid single cloud lock-in). This means the databases chosen must have equivalents on different clouds. The easiest way to satisfy that is to choose databases that are offered on all clouds (like Postgres, MySQL, Mongo). For example, if you build on PostgreSQL, AWS, Azure, GCP all have managed PostgreSQL options. If you built on something bespoke like Google Spanner, you’d fail that multi-cloud requirement since only Google has Spanner. Thus the RFP indirectly forces picking non-bespoke, widely supported tech. This is another way RFPs focusing on interface/pattern indirectly kill the incentive to use a one-of-a-kind database technology.

In summary, **procurement language is shifting to require standards and patterns rather than specific engines**, further encouraging the use of general-purpose, widely compatible databases. This trend reduces the prevalence of truly unique, one-off databases in deployed solutions because it’s harder to justify them when you’ve committed to “must be replaceable or standardized.” It’s a self-reinforcing cycle: as multi-model databases become more capable, they meet more requirements; as more requirements demand standard interfaces, multi-model solutions get chosen more; specialist DBs that don’t align with a standard API struggle to be included.

Counter-Evidence: The Enduring Role of Specialized Databases

While the prevailing trend points toward a fading era of bespoke databases, it is important to acknowledge counter-evidence and scenarios where specialized databases continue to thrive or new ones emerge. The database ecosystem is diverse, and **“one size fits all” can still be an anti-pattern in certain situations**. Here we examine the arguments and evidence suggesting that specialized, purpose-built databases will not disappear entirely – and might even see resurgences for particular needs – despite the multi-model consolidation trend.

1. Performance and Scale Extremes Preserve Niches: Even as general-purpose databases narrow the performance gap, there remain extreme cases where specialists significantly outperform generalists. High-frequency trading systems, for example, might still rely on in-memory, specialized tick databases or custom time-series solutions to achieve microsecond latencies that a general DB can't reach. The Oso analysis of vector databases conceded that *"standalone vector databases"* could differentiate via proprietary optimizations (like custom embeddings or faster reranking) that general platforms with "sprawling priorities" might not match . They gave the example that **Chroma (a specialized in-memory vector DB) was significantly faster (40–80 ms queries) than Postgres pgvector** – not due to a fundamentally different algorithm, but by virtue of being entirely in-memory and optimized for that use . This suggests that for use cases where every millisecond counts or where working set cannot fit in memory on a single node, a specialized distributed or in-memory engine still wins. The general DB could be made to perform similarly (by putting it on an in-memory optimized instance), but at some point the complexity or cost of pushing a general system to those extremes outweighs using a purpose-built solution. For instance, if you truly need sub-50ms similarity search on billions of vectors globally, a dedicated vector DB with distributed indexing might be more straightforward than sharding Postgres and managing that. Likewise, even though MySQL HeatWave brings analytics into MySQL, for a 100 TB data warehouse with complex BI workloads, Snowflake or Redshift might still substantially outperform a MySQL-based solution and offer more mature tooling. In short, *specialized databases continue to dominate in the "ultra-high-end" of scale or performance*, which means organizations at that tier will still adopt them.

2. Functional Specialization and Developer Productivity: Performance aside, specialist databases often provide query languages or data models that are more *convenient* for certain tasks. Graph databases are a key example. Developers modeling highly connected data sometimes find graph query languages (Cypher, Gremlin, GraphQL) far more natural and productive than SQL with join tables. While multi-model databases can store graph data (e.g., using edge tables in Postgres), the developer experience might not be as good. Neo4j, for instance, offers "index-free adjacency" and a traversal engine that makes graph algorithms straightforward . One InfoQ panelist remarked that *"in a native graph database, we can traverse millions of relationships per second…we think of this as query-driven modeling that is open to domain experts rather than just DB specialists"* . That level of specialization in the data model (and the ease of expressing graph queries) is hard for a general DB to replicate fully without essentially embedding a graph engine. Indeed, multi-model platforms that try to add every model risk complexity or conflict; an InfoQ expert cautioned *"I am fearful of multi-model databases ending up conflicting with their own feature sets and not being really good at anything"* . There is anecdotal evidence that some teams tried to use a multi-model database in "graph mode" or "JSON mode" and found it less capable than a dedicated system. For instance, developers might try to use a general DB for full-text search, only to hit limitations and revert to Elastic because of richer querying or scaling. **This suggests specialized DBs can endure where they still provide a significantly improved developer experience or feature set.** We see this with *search engines, graph DBs, time-series analytics UIs, etc.* – Elastic's Kibana

ecosystem, Neo4j's graph algorithms library, etc., can be more feature-rich for their domain than generic tools on a multi-model DB.

3. Multi-Model Trade-offs and Complexity: Operating a multi-model database that does "everything" may introduce its own complexity. Perry Krug in the InfoQ panel pointed out that multi-model can become "not good at anything" if stretched too far, and that legacy NoSQL users often preferred polyglot because "each technology excelled at a limited set of use cases" . Jim Webber mentioned *"the jury is out on multi-model…we saw in the era of RDBMS when we shoehorned everything into relational – that spawned NoSQL!"* . This historical reminder indicates a pendulum: if multi-model DBs begin to compromise too much (e.g., performance trade-offs or overly complex configuration to handle different models), architects might again opt for specialized systems for clarity and efficiency. A multi-model DB might also have *higher resource usage or cost* if it has to run heavier components for each model. In cloud environments where you pay for what you use, it might be cheaper to use a slim purpose-built DB for a component than to scale up one big multi-model cluster to handle all needs.

4. Vendor and Open-Source Ecosystem Moves: We should note that not all vendors are embracing convergence – some explicitly advocate *purpose-built databases for each workload*. Amazon, for instance, in its documentation, still champions having **"eight purpose-built databases"** for different use cases so customers "don't have to compromise as often happens with converged databases" . They provide a relational, key-value, document, graph, time-series, etc., each separate . This is a strategic choice: AWS believes (and markets) that using the right tool yields optimal performance and that its cloud makes it easier to manage many tools. AWS's Prescriptive Guidance suggests that modern applications with microservices *"require specific database types to support each workload"* , emphasizing no need to compromise. This philosophy is essentially *against* the single multi-model approach. As long as a major player like AWS pushes this, many customers (especially large ones with diverse needs) will follow that guidance and continue using multiple specialized engines in their architecture. In effect, AWS's stance is counter-evidence to "bespoke DBs fading" – they argue for continued (managed) polyglot persistence. And given AWS's market share and influence, their strategy could prolong the life of specialized DB categories. For example, rather than adding time-series to Aurora, AWS made Timestream; rather than adding graph to DynamoDB, they have Neptune. They tout that *"no need to compromise as often happens with converged DBs"* – implying that converged solutions inherently involve trade-offs . There is truth here: a converged engine's complexity might mean it's not the absolute best at any one thing (though it's good at many). If a use case is critical enough, some will prefer best-of-breed specialist. So long as multiple DBs can be managed (and cloud makes it easier), that approach stays valid.

5. New Data Modalities Could Spawn New Specialists: Looking ahead, as new technologies and data paradigms arise, they often come with new specialized databases. We saw this with *blockchain and ledger databases* – AWS QLDB and others popped up to handle immutable ledger use cases with cryptographic verification, which traditional databases didn't offer out of the box. Quantum computing or homomorphic encryption might lead to new forms of data storage with very specialized requirements. It's plausible that by 2030, we'll see new "bespoke" databases for, say, real-time immersive data for AR/VR, or for decentralized data networks, etc.

These might start as niche bespoke systems because existing DBs aren't built for those patterns. Eventually they might merge into mainstream, but there's a cycle: new needs create new bespoke solutions, which then over time get integrated. For example, vector databases were new in 2020s and got partially integrated; in late 2020s maybe real-time AI agent state stores or feature stores are a new category that initially separate. So the "era of bespoke DBs fades" might hold for current workloads, but *entirely new workloads* can re-introduce specialization at least temporarily until integration catches up.

6. Legacy Systems and Inertia: Another reason bespoke databases won't vanish overnight is simply the installed base of legacy systems and the inertia in large enterprises. Many companies have, say, a decades-old IBM IMS (hierarchical DB) or a highly tuned Oracle RAC for their ERP. They are not going to switch to a multi-model just because the trend says so – if it works and the cost of change is high, they stick with it. So in the aggregate market, specialized databases (including legacy ones) will continue to hold a share. The DBMS market is huge and changes slowly; DB-Engines ranking still shows dozens of specialized systems being used. They won't all evaporate by 2030. Even if no new competitor enters a category, existing ones can remain. For example, mainframe databases (IMS, Adabas) or even Microsoft Access in small businesses – these are specialized in their context and likely still around.

7. Multi-Model Not Always the Right Choice for Teams: Polyglot persistence is sometimes still the optimal architecture. Tim Berglund on InfoQ commented that he was *"never too persuaded of polyglot persistence as an architectural strategy"* in greenfield because of complexity , but even he acknowledged *"real use cases exist for different models… an architect might prefer a graph here, SQL there, a key-value store for another part"*, and that multi-model is just one way to address that challenge . Another panelist (Jim Webber) maintained *"polyglot has strong credibility"* because microservices can each choose the right DB for their service . This microservice architecture trend means in a large application, each bounded context could pick a DB tech that suits it best. If managed well, the operational burden is mitigated by each service being small. And crucially, microservices decouple teams – one team might use a graph DB if it suits their service, while another uses Postgres. This organizational factor can sustain variety in DB choices rather than a single multi-model for all. In fact, *some companies intentionally encourage using the best DB per service to optimize each independently.* The trade-off is complexity, but if they have platform engineering to handle automation, it can work. So while multi-model DBs attempt to simplify by consolidation, microservices architecture can tolerate or even prefer multiple stores, which argues bespoke DBs will continue to be used where beneficial.

8. Use-case Examples of Specialist Success: We can point to specific domains:

- **Time-series (industrial)**: Even though Timescale/Postgres is popular, many IoT deployments still choose InfluxDB or Kafka + Druid for handling millions of sensor events per second because those specialized pipelines are proven at scale. For example, some large monitoring systems (like for telecommunications or power grids) might accumulate such volume that an embedded time-series engine's overhead in Postgres is too high. Timescale has limits (it is essentially one big Postgres instance scaled up, or distributed

in a newer commercial version but that's relatively new). InfluxDB OSS and others can cluster/shard too. So for extreme throughput, a specialized TSDB can still be the better choice.

- **Edge and Embedded**: In edge computing or mobile, a very lightweight specialized database (like SQLite, or a tiny TSDB for IoT device) might be needed. Multi-model engines are often heavier. So at the edge, bespoke might remain (SQLite is technically multi-model? not really, just relational – but it's specialized for embedded use).

- **Analytical engines**: The big data world keeps innovating specialized engines (Spark, Flink for streaming, etc.). While Snowflake and others try to be multi-model (introducing some JSON, some Python execution, etc.), there's still a lot of niche engines for various analytics (graph analytics engines, time-series analytics frameworks). These might not call themselves "databases" but compete in that space for specific analysis tasks.

9. Maturity and Support: Not to be overlooked, some specialized databases have decades of maturity for their niche. For instance, IBM Db2 on mainframe or Oracle for high-end OLTP – a new multi-model alternative might not match the rock-solid stability and support, so risk-averse enterprises stick with them. Similarly, Neo4j as a graph DB has tuned its performance for deep traversals; a multi-model adding graph might be relatively immature and have edge-case bugs. Companies valuing that maturity will stick to the proven specialist.

10. Human Factors: DBAs and data engineers often specialize in certain technologies. If an organization has a team deeply skilled in MongoDB and it serves them well, they might resist migrating to an all-in-one system even if technically feasible. There's investment in training and tooling around each DB. This human factor can slow the fade of bespoke DBs – people trust what they know.

Counter-Evidence Summary: Essentially, the counterpoints highlight that *"bespoke" databases won't disappear; rather, the centre of gravity may shift but fringes and high-end use cases keep them around.* A realistic future likely involves a mix: many general-purpose, multi-model databases handling the bulk of workloads (especially in the cloud, and for new applications with moderate requirements), while specialized systems continue to exist and even be preferred for *specific scenarios that demand their unique strengths*. We can foresee something like: multi-model DBs capture an increasing share of new deployments (especially in SMBs and standard enterprise IT), but in domains like ultra-scale analytics, extremely low-latency workloads, or specialized functional domains, purpose-built databases retain a significant role.

In conclusion, **the era of bespoke databases is evolving, not completely ending.** The trend is toward fewer, more capable platforms – but history suggests that whenever one wave of consolidation happens, new data challenges emerge that start the cycle of specialization again. As one expert cleverly noted, *"it's hard to swallow that any single database can be all things to all people… that learning [of limits] was what spawned NoSQL!"* . That cautionary note reminds us that the industry must be careful not to repeat the mistakes of past monocultures. It implies

that even as multi-model solutions become dominant, we should expect and even welcome specialized databases where they make sense. The key will be interoperability – ensuring those specialists can plug into the broader ecosystem without locking users in. And indeed, with trends like standard APIs and cloud services, we may see a scenario where *specialized engines exist "under the hood" of multi-model services or as easily-swappable managed services*, thus giving the best of both worlds.

The counter-evidence, therefore, doesn't refute the consolidation trend but rather nuances it: bespoke databases will **fade in prominence for general use** but remain **vital in particular niches and at the cutting edge of performance**. Enterprises will likely approach architecture with a "90/10" rule – try to cover 90% of needs with a general solution, but don't hesitate to employ a specialized one for the 10% where it really matters. This balanced view aligns with how technology waves usually pan out, ensuring that innovation in specialized data management continues even as mainstream usage consolidates.

Conclusion

The database landscape from 2026 through 2030 is poised to be markedly different from the explosion of bespoke, single-purpose databases that characterized the previous two decades. **Multi-model and general-purpose databases are on track to capture the majority of new use cases**, heralding a consolidation around a few flexible platforms rather than a polyglot patchwork. We have seen that virtually every major database provider – from open-source communities to cloud vendors – is investing in broadening their systems' capabilities: relational databases now store documents, run analytics, handle time-series and vectors; cloud services present themselves as drop-in replacements for multiple traditional databases under one umbrella.

By 2030, we expect a typical enterprise's data architecture to use fewer distinct database technologies than in 2020, with an emphasis on those that can do more than one thing well. **Time-series and vector databases are well on their way to becoming features of larger systems** rather than standalone products for most users, as evidenced by PostgreSQL and MySQL absorbing those workloads . **Cloud database offerings will continue to emphasize open compatibility**, resulting in a market where the API (Postgres, MySQL, etc.) matters more than the engine brand – a trend already visible with services named for their compatibility (e.g. Amazon DocumentDB with MongoDB API) . **Protocol-compatible alternatives have proliferated**, meaning organizations can swap the engine beneath their applications with minimal disruption (illustrated by MySQL's forks and reimplementations, and the rise of alternatives like Scylla for Cassandra) . **General-purpose databases have aggressively closed performance gaps** on specialists in many domains, often "good enough" such that maintaining a separate specialized system is no longer worth the incremental gain . Finally, **procurement practices are reinforcing these trends** by mandating standards and flexibility – RFPs increasingly specify requirements in terms of capabilities and interfaces (SQL, API compatibility) rather than tying solutions to specific proprietary databases .

However, as our counter-evidence discussion emphasized, this does not spell the absolute end of specialized databases. They will persist and excel in niches where they are truly needed – often behind the scenes, integrated into broader platforms or deployed for targeted high-scale tasks. The likely outcome is a healthy middle ground: a smaller set of powerful, multi-model databases covers the bulk of applications (simplifying the lives of developers and DBAs), while a cohort of specialized systems continue to innovate on the frontier of performance and new data paradigms. In essence, **the era of "one new database for every new problem" is fading, but the ethos of using the right tool for critical jobs remains valid**. Organizations will increasingly choose databases by asking "can our primary database platform handle this well enough?" and only if the answer is a firm no will they seek a niche solution.

For technical decision-makers, this evolution means that skills in a few versatile database technologies (especially PostgreSQL, given its ubiquity and extensibility) will go a long way, and designing systems with portability in mind is becoming easier. It also means that architectural focus can shift from plumbing between dozens of datastores to higher-level concerns, as fewer moving parts are needed for a given project. **By 2030, we expect RFPs and system designs to routinely list requirements like "must support PostgreSQL interface, JSON and graph queries, full-text search, and time-series aggregations" – things that would have implied five different products in 2015, but can be satisfied by one or two in 2030.**

In sum, the industry is moving towards consolidation around multi-model, multi-workload databases not because of hype, but because of genuine improvements in those platforms and a maturation of understanding that integration overhead is costly. The share of new databases that are single-purpose will diminish as vendors continue to roll multiple capabilities into their flagship systems and as cloud services make switching engines behind a familiar interface trivial. **The bespoke database era – defined by a proliferation of highly specialized, standalone DBMS – is indeed fading for mainstream IT.** Its legacy, however, lives on in the rich features that have been absorbed into today's general-purpose databases, and in the specialized systems that still power the most demanding applications. Savvy organizations will leverage these generalists for breadth and ease, while still keeping an eye on specialists for the rare cases where they provide a decisive edge.

Ultimately, the trajectory suggests a future where **database choice is less about name and more about capability** – an ecosystem where one can largely "speak in APIs and patterns" and trust that multiple competing implementations (cloud or open-source) can fulfill those needs. That represents a significant maturation of the data management field. Technical leaders should prepare for this future by investing in technologies and skills that align with widely adopted interfaces and by architecting systems to be modular at the API level. The freedom to swap out or consolidate databases without disrupting business logic will be a competitive advantage. Those who insist on clinging to bespoke engines for non-compelling reasons may find themselves on the wrong side of history (and budget approvals), as the momentum is clearly towards simplification, standardization, and multi-model versatility in the database tier.

Footnotes (Chicago Style):

1. Rapydo (2025). *Relational Databases in the Near and Far Future*. *Rapydo Blog*. – Citing Gartner research: as of 2024, ~61% of database market is cloud, 91% of new DB growth is cloud .

2. Rapydo (2025). *Rise of Multi-Model Databases in Modern Architectures*. – Multi-model DBs support multiple data models in one platform, eliminating polyglot complexity; adopters report consolidation and efficiency gains .

3. Akmal Chaudhri (2022). “SingleStoreDB: The Rise and Rise of Multi-Model Database Systems.” *Medium (via DB-Engines)*. – Relational vendors extended their products (JSON, time-series, etc.) to meet new requirements, ushering in multi-model era .

4. InfoQ (2019). “Virtual Panel: Current State of NoSQL Databases.” – Discussion on multi-model vs polyglot: experts warn one DB might not fit all; polyglot can still be credible for microservices .

5. Oso (2025). “Feature or Product?: Vector Databases” – Observes many companies used Postgres pgvector or MongoDB for vector search rather than new vector DBs; Postgres/Mongo had double the adoption of specialized vector DBs in a survey . Also notes pgvector performance competitive with Pinecone .

6. Rapydo (2025). *Relational Databases in the Near and Far Future*. – Relational DBs (MySQL, Postgres) evolved: now support JSON, horizontal scaling, HTAP analytics, and even AI/ML workloads (via vectors) that gave rise to “vector databases,” with relational “responding too” via extensions like pgvector .

7. TheCUBE/Wikibon (2022). “Oracle MySQL HeatWave…”. – MySQL has ~20.6% of transactional DB market (~190k customers) not counting forks (MariaDB, Percona) or cloud MySQL services . Implies actual MySQL protocol use is higher when including those. Also highlights MySQL’s lack of analytics historically and Oracle’s HeatWave adding analytics in MySQL to eliminate ETL .

8. Abhishek Tiwari (2015). “Rethinking the database with drop-in replacements.” – Introduced concept of drop-in replacements like Amazon Aurora (MySQL-compatible with 5× throughput) and Scylla (Cassandra-compatible in C++). Aurora described as game-changer beyond earlier forks MariaDB/Percona ; Scylla as fully Cassandra API-compatible replacement with much higher performance .

9. AWS Prescriptive Guidance (2021). “Purpose-built versus converged databases.” – AWS’s viewpoint: modern apps often use specific DBs for specific microservices, and that purpose-built DBs avoid compromises of converged approach . Table of AWS offerings shows distinct services for relational, key-value, document, graph, time-series, etc., each optimized .

10. HashiCorp Terraform Enterprise Requirements (2023). – Requires a PostgreSQL database and explicitly allows *“a PostgreSQL-compatible server such as Amazon Aurora PostgreSQL”* , underscoring that compatibility can fulfill the requirement interchangeably with the named database.