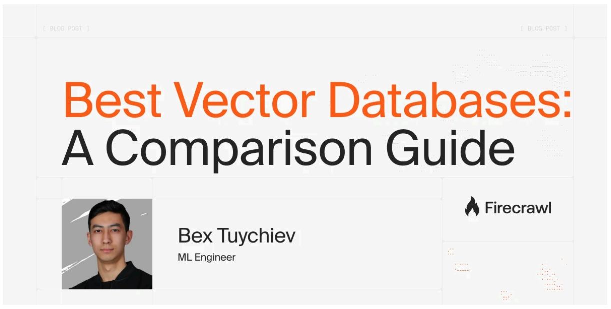
Introducing Spark 1 Pro and Spark 1 Mini models in /agent. Try it now →

‹ Blog

Best Vector Databases in 2025: A Complete Comparison Guide

Bex Tuychiev Oct 09, 2025



Firecrawl

application and AI application development. You search and find more than a dozen options: Pinecone, Milvus, Qdrant, Weaviate, ChromaDB, pgvector, and more. Each vendor claims to be the fastest, most scalable, most developer-friendly solution. The benchmarks contradict each other. And you still need to decide.

This vector database selection guide gives you what the vendor sites won't: real performance numbers from **VectorDBBench**, honest trade-offs between managed and self-hosted options, and a framework for choosing the right vector database for your needs. You'll learn which solution fits your scale, use case, and existing infrastructure. No hand-waving, no vendor pitches. Just the information you need to decide and move forward.

What you'll find here:

- **Evaluation framework** for comparing databases
- **Honest comparison** of 14 major databases
- **Decision matrices** by scale, use case, and infrastructure
- **Data pipeline integration** with Firecrawl

Quick navigation: If you already know how to evaluate vector databases, skip to **the database comparison** or jump to **the decision framework**.

How to Evaluate Vector Databases

Vector Database Tutorial: Getting Started with the Fundamentals

Before comparing specific databases, you need to understand what actually matters: recall trade-offs, architectural differences, and performance metrics that reflect the reality of deploying production AI systems.

Skip to ***The Vector Database Landscape*** *if you already understand recall/precision trade-offs, HNSW architecture, and production performance metrics (p99 latency, concurrent QPS).*

What is a vector database?

A vector database stores and searches data as high-dimensional vectors (arrays of numbers that represent meaning). When you convert text, images, or other content into embeddings using AI models, you get vectors that capture semantic meaning. Vector databases let you find similar items by measuring the distance between these vectors.

Instead of exact keyword matches, you search by meaning through semantic similarity. This powers modern AI applications: chatbots that retrieve relevant context, recommendation systems that understand user preferences, and search engines that grasp intent. Vector databases enable RAG (Retrieval Augmented Generation), power ChatGPT-style applications, and serve as the knowledge base layer for LLM integrations.

The core trade-off: Recall vs speed

Vector databases make a core compromise. Exact nearest neighbor search (checking every vector to find the true closest matches) is too slow for LLM applications in production, so they use vector indexing algorithms for approximate search. The question is how much accuracy you sacrifice for speed.

A system running at 95% recall (successfully retrieving 95 out of every 100 relevant documents) misses 1 in 20 relevant documents. At 99% recall, you miss 1 in 100. That difference determines whether your RAG (Retrieval Augmented Generation) system regularly misses critical context or almost never does.

Performance benchmarks only mean something with a recall number attached. Comparing “10ms at 90% recall” to “50ms at 99% recall” is meaningless because they operate at different recall levels and solve different problems.

Architecture: Purpose-built vs extensions

Different architectures approach this recall/speed trade-off differently.

Purpose-built databases like Pinecone, Milvus, Qdrant, and Weaviate use vector-optimized storage engines, query planners, and index structures. They implement **HNSW (Hierarchical Navigable Small World)**, a graph-based algorithm that searches vectors by navigating through multiple layers from coarse to fine approximations. This handles billions of vectors well because algorithm complexity grows logarithmically, not linearly, regardless of vector dimensionality.

Extensions like `` `pgvector` ``, Redis, MongoDB, and Elasticsearch add vector indexes to existing storage engines. You keep vectors and relational data in one system, query them in the same transaction, and avoid managing separate infrastructure. Beyond 50-100M vectors, extensions hit throughput and latency limits that purpose-built systems avoid.

Recent benchmarks challenge this assumption. `` `pgvectorscale` `` achieves **471 QPS (Queries Per Second) at 99% recall on 50M vectors**. That’s 11.4x better than Qdrant’s 41 QPS at the same recall (May 2025 benchmarks). This shows extensions can compete with specialized systems at moderate scale.

Choose purpose-built if vectors are your primary workload at hundreds of millions scale. Choose extensions if you already run PostgreSQL, Elasticsearch, MongoDB, or Redis, especially if you need vectors alongside traditional data.

Performance: What actually matters

Architecture determines potential performance, but specific metrics show real-world behavior.

Latency p99 (99th percentile, measuring the slowest 1% of queries) matters more than median. Slow tail queries degrade user experience more than good medians improve it. A system with 10ms median and 500ms p99 feels slower than 20ms median and 50ms p99.

Concurrent throughput (QPS) shows behavior under actual load. A database handling 100 simultaneous queries at 30ms beats one that does 1 query at 10ms but drops to 200ms under load. Single-query benchmarks lie about production performance.

If you're updating vectors frequently in production, write performance determines whether you can keep up. Systems vary from thousands to over 160K vectors per second per node at billion-scale, depending on configuration and embedding precision requirements. For applications retrieving large result sets (k > 10,000), batch retrieval efficiency varies significantly between databases and becomes a critical factor.

Quantization (compressing vectors by using fewer bits per dimension, like reducing 32-bit floats to 8-bit integers) reduces memory costs with minimal recall impact. Reducing 32-bit floats to 8-bit integers **cuts memory 75% while**

maintaining high accuracy. Test which quantization methods work with your embedding model and acceptable recall thresholds.

Features and filtering approaches

Beyond raw performance, feature implementation determines whether a database fits your use case.

Hybrid search (combining vector similarity, keyword search, and metadata filters in a single query) gives more precise results than vector search alone. Weaviate and Qdrant include this without plugins or configuration. Filtering implementation determines whether this works at scale.

When you filter vectors, the approach matters. Pre-filtering applies filters before vector search, which is faster but disrupts HNSW graph traversal and can reduce recall. Post-filtering searches first then removes non-matching results, which maintains recall but scans more vectors. Test with your filter selectivity. Either approach works under 10M vectors, but implementation matters beyond that scale.

Most RAG and knowledge base applications need vector search, metadata filtering, and steady writes. Multi-modal support (text, images, audio, video) and real-time updates matter for specific use cases.

Cost and operational reality

At scale, you typically optimize for two of three: performance, cost, or simplicity.

Managed services trade dollars for time. Pinecone uses usage-based pricing (storage + read/write units) and handles scaling, monitoring, and updates. Actual costs for 10M vectors vary from under $100/month for low-traffic applications to $2,000+ for high-throughput production workloads.

Self-hosted costs less in hosting fees but needs expertise. Milvus is free, but you need engineers who can configure HNSW parameters, debug distributed systems when they fail, and optimize query performance. Engineering time, monitoring, and backups add hidden costs.

Infrastructure integration often determines total cost more than database pricing. If you run PostgreSQL, adding `pgvector` costs only your existing PostgreSQL infrastructure. You don't need an additional database to run. That's usually cheaper than managed services, even if they're faster.

Making the decision

Most databases handle sub-100ms queries under 10 million vectors. Differences appear at 50M and become critical beyond 100M. Unless you have concrete scaling plans within 6 months, solve for current scale.

Decision framework by infrastructure and scale:

Your Situation	Recommendation
Existing PostgreSQL (<50M vectors)	Start with `pgvector` + `pgvectorscale`
Existing Elasticsearch/MongoDB/Redis	Add vector search to existing stack
New project (<10M vectors)	Qdrant Cloud (best free tier) or ChromaDB (prototyping)
New project (10-100M vectors)	Pinecone (easiest), Weaviate (hybrid search), or Milvus (cost-effective if self-hosted)
New project (>100M vectors)	Milvus/Zilliz Cloud or Pinecone serverless

Decision framework by use case:

Use Case	Recommended Databases	Why
RAG applications	Weaviate, Pinecone, Qdrant	Sub-100ms queries, hybrid search, excellent metadata filtering
Multi-modal search (text + images + video)	Marqo, Weaviate, Qdrant	Native multi-modal support, unified embedding handling

Use Case	Recommended Databases	Why
Real-time updates	DataStax Astra, Elasticsearch, MongoDB	Immediate consistency, operational workload optimization
Edge/on-device deployment	ChromaDB, Qdrant, Weaviate	Embedded modes, compact footprint, resource efficiency

Tools like **VectorDBBench** provide standardized baseline comparisons across 15+ databases with consistent datasets and metrics. Use it to run your own benchmarks with your production workload, query patterns, and hardware configuration. The right database fits your constraints, not benchmark leaderboards.

The Vector Database Landscape

The vector database market has grown from $1.73 billion in 2024 to a projected $10.6 billion by 2032, reflecting the rapid adoption of RAG and semantic search in production applications. Open-source adoption has accelerated alongside this growth: Milvus leads with over 35,000 GitHub stars, followed by Qdrant (9,000+), Weaviate (8,000+), and ChromaDB (6,000+).

We evaluated 14 major vector databases across managed, open-source, and extension categories. Based on popularity (GitHub stars, market share), versatility (range of use cases), production maturity (proven deployments), and cost-effectiveness, these 6 databases handle 80% of vector database needs. For specialized requirements, we cover 8 additional databases that excel in specific scenarios.

Quick-scan comparison table

The sections below give an in-depth coverage of key vector databases, so are heavy in detail. If you want to spare yourself from that, here is a table that accurately captures the key pieces of information:

Top 6: Most Use Cases

Database	Type	Best For	Key Strength	Key Limitation	Pricing
Pinecone	Managed	10M-100M+ vectors, zero ops	Easiest, serverless	Usage-based pricing	Free tier | Paid: usag based ($0.33/GE ops)
Milvus	Open-source	Billions, self-hosted	Most popular OSS	Ops complexity	Free (infra costs) | Managed: $99/mo+
Weaviate	OSS + Managed	RAG <50M, hybrid search	Best hybrid search	14-day trial limit	Free (OSS Cloud: $25/mo (after trial)

Database	Type	Best For	Key Strength	Key Limitation	Pricing
Qdrant	OSS + Managed	<50M, filtering	Best free tier (1GB)	Lower throughput >10M	Free tier: 1GB forev Paid: $25/mo
pgvector	PostgreSQL ext	PostgreSQL users	Unified data model	Not for >100M	Free (PostgreS infra only)
Elasticsearch	Search + vector	Existing Elastic users	"Just works" reliability	Higher latency	Free (OSS Elastic Cloud vari

Specialized Options

Database	Type	Best For	Key Strength	Key Limitation	Pri
ChromaDB	OSS (embedded)	Prototypes \ <10M	Best DX	Not for scale	Fre Cl cre
Redis	In-memory ext	Low-latency <10M	Sub-ms latency	Memory-bound	Cl pri (m co
FAISS	Library	Billions, custom	Meta-proven, GPU	Requires integration	Fre (lib
Marqo	Open-source	Multi-modal	Text+images+video	Newer/less proven	Pri str en
DataStax Astra	Managed (Cassandra)	Real-time updates	Immediate updates	Cassandra expertise	En (cc sa
MongoDB	Document + vector	MongoDB users	Vectors+documents	Not for pure vector	Atl $5 70
Zilliz Cloud	Managed (Milvus)	100M-1B+ enterprise	Massive scale	Steeper curve	Fre | P

Database	Type	Best For	Key Strength	Key Limitation	Pri
					$9
SingleStore	Analytics + vector	Real-time analytics	Analytics+vectors	Opaque pricing	En (co sa

If you decide to read on, good luck!

Top 6 Databases for Most Use Cases

These 6 databases cover the vast majority of vector database deployments. They balance performance, cost, developer experience, and production maturity. Unless you have specialized requirements (multi-modal search, extreme scale, specific infrastructure), start your evaluation here.

Pinecone

The managed solution for teams who want production-ready vector search without operational overhead

Type	Fully managed, serverless
Performance	7ms p99 latency, auto-scaling, proven at billions of vectors

Pricing	Usage-based ($0.33/GB storage + read/write operations), free tier available
Choose when	Building commercial AI products, want zero ops, need SLA guarantees
Avoid when	Tight budget, need infrastructure control, have in-house DB expertise

Pinecone solves a specific problem: you want production-grade vector search without managing infrastructure. If you've ever run a database in production, you know the operational burden. Monitoring, scaling, updates, backups, debugging performance issues at 3am. Pinecone removes all of that.

Here's how the managed approach works. You create an index, upload vectors, and query. Pinecone handles scaling automatically. Traffic spikes from 100 queries per second to 10,000? The system adjusts without you touching configuration. Your dataset grows from 10 million to 100 million vectors? Same thing. This isn't unique to Pinecone, but they've refined it to the point where it works reliably.

Performance backs up the convenience. Tests show 7ms p99 latency, which means 99% of your queries return in under 7 milliseconds. Compare that to Elasticsearch's 1600ms for exact kNN in the same benchmark. The serverless tier auto-scales, so you don't provision capacity. You use what you need, pay for what you use.

The trade-off is cost. Pinecone uses usage-based pricing with separate charges for storage ($0.33/GB/month), read operations, and write operations. For large deployments, this adds up quickly. Self-hosted alternatives like Milvus or `` `pgvector` `` cost a fraction of that, but you manage them yourself.

Vendor lock-in is real. Pinecone uses a proprietary system. Migrating to another database means rebuilding your integration. The API is well-documented and the SDKs work reliably, but you're committed once you build on it. Some teams accept this trade-off for the operational simplicity. Others don't.

The integration ecosystem is mature. LangChain and LlamaIndex both have first-class Pinecone support for building RAG applications. Most **RAG frameworks** and AI tools integrate without friction. Documentation is comprehensive, support responds quickly, and the system is stable. These aren't flashy features, but they matter when you're shipping to production.

You choose Pinecone when time-to-market matters more than cost optimization. Startups building AI SaaS products, teams without database expertise, companies that value reliability over infrastructure control. You avoid it when budget is tight, when you need full control over your stack, or when you already run self-hosted databases and have the expertise to manage another one.

Milvus

Open-source powerhouse for billion-scale deployments when you have engineering resources and need cost efficiency

Type	Open-source, cloud-native, self-hosted or managed via Zilliz
Performance	Low single-digit ms latency, <30ms p95, proven at billions of vectors
Pricing	Free (Apache 2.0), infrastructure costs only, managed via Zilliz ($99/month+)
Choose when	Billion-scale needs, have ops expertise, cost-sensitive, want control
Avoid when	Small team, need managed simplicity, <10M vectors, want fastest deployment

If Pinecone's managed approach doesn't fit your budget or you need more control, Milvus is the most popular open-source alternative, with over 35,000 GitHub stars. That popularity reflects real production usage. Teams running billions of vectors choose Milvus because it's built for that scale and costs nothing except infrastructure.

The architecture splits storage and compute, letting you scale indexing separately from queries. Add nodes when you need more throughput, remove them when traffic drops. The HNSW index implementation handles billions of vectors with **low single-digit millisecond latency** and sub-30ms p95 at millions of vectors. Performance holds up because the system is designed for distributed workloads.

But Milvus requires operational expertise. You manage Kubernetes deployments, configure index parameters, debug distributed systems when things break. The learning curve is real. Teams without data engineering experience struggle with it. You need people who understand vector indexing trade-offs, can tune HNSW parameters for your workload, and know how to troubleshoot performance issues in distributed systems.

The cost advantage is significant if you have that expertise. Self-hosting Milvus on AWS costs maybe $500-1000 monthly for infrastructure that handles 50 million vectors. Compare that to Pinecone's $3,500 for the same scale. The difference compounds as you grow. At billions of vectors, self-hosted Milvus saves tens of thousands monthly compared to managed alternatives.

Integration quality varies. The Python SDK works well, documentation covers most use cases, and the community is active on GitHub and Discord. LangChain and LlamaIndex both support Milvus. But you'll hit edge cases that aren't documented, and solving them requires reading source code or asking in community channels. That's normal for open-source, but it's different from Pinecone's support experience.

Zilliz Cloud offers managed Milvus if you want the architecture without the operations. It's more expensive than self-hosting but cheaper than Pinecone, and you get enterprise features like better observability and support. That's the middle ground.

Best fit for organizations building at billions of vectors with engineering resources to manage distributed systems. Large tech companies and data-intensive startups use Milvus where infrastructure savings (often 70%+ vs managed alternatives) justify the operational complexity. Small teams, projects under 10 million vectors, or organizations needing managed simplicity should look elsewhere.

Weaviate

The hybrid search specialist combining vectors, keywords, and filters with exceptional documentation

Type	Open-source with managed option, hybrid search focused
Performance	Sub-100ms for RAG, ~50ms on 768-dim embeddings, GraphQL API

Pricing	OSS free, Cloud starts $25/month (dimension-based pricing) after 14-day trial
Choose when	Need hybrid search, building POCs (excellent docs), want modularity
Avoid when	Need max vector performance, very tight budget, >100M scale, prefer REST

While Pinecone and Milvus focus on pure vector search, **Weaviate** does one thing better than any other database in this comparison: hybrid search. If you need to combine vector similarity, keyword matching, and metadata filtering in a single query, Weaviate handles it natively and well. That's why teams building RAG applications choose it.

Here's how the hybrid search works. You query with a vector embedding, add keyword filters using BM25 (a ranking algorithm for full-text search), and apply metadata constraints. Weaviate processes all three simultaneously and returns ranked results. Other databases add these features separately or require you to combine separate queries. Weaviate builds it into the core architecture. The **GraphQL API** exposes this cleanly, though some developers prefer REST.

Documentation is exceptional. The tutorials are clear, examples work out of the box, and concepts are explained well. This makes Weaviate ideal for proof-of-concept projects. You can build a working RAG system in an afternoon by following the guides. The modular architecture lets you plug in different embedding models, vectorizers, and rerankers without rebuilding your application. That flexibility matters when you're experimenting.

Performance is good but not the absolute best for pure vector operations. Query times stay **sub-100ms for most RAG workflows**, which is fast enough for user-facing applications. But in benchmarks focused purely on vector similarity at massive scale, purpose-built systems like Milvus or Pinecone edge ahead. The graph features that enable hybrid search add some overhead. That's the trade-off.

Resource usage becomes a concern above 100 million vectors. Teams report that Weaviate needs more memory and compute than alternatives at very large scale. Below 50 million vectors, it runs efficiently. Beyond that, you need to plan capacity carefully. The hybrid search features are powerful, but they cost something in resource consumption.

Pricing for Weaviate Cloud starts at $25 monthly after a 14-day trial. That's competitive for managed services, but the trial period is the shortest among major options. Qdrant gives you 1GB free forever, Pinecone has a free tier, but Weaviate requires payment after two weeks. For production deployments this doesn't matter much, but it affects prototyping budgets.

The open-source version works well for self-hosting. You can run Weaviate in Docker or Kubernetes, manage it yourself, and pay only infrastructure costs. The deployment is straightforward compared to Milvus. Both managed and self-hosted options give you the same hybrid search capabilities.

Hybrid search requirements make Weaviate a natural choice. The exceptional documentation and modular architecture work well for proof-of-concept projects. Both open-source and managed deployment options provide flexibility. Performance scales efficiently to 50 million vectors, though resource requirements increase beyond 100 million. The 14-day trial period and GraphQL-first API won't suit every team, but for hybrid search use cases, Weaviate delivers.

Qdrant

Budget-friendly vector database with the best free tier and excellent filtering for moderate-scale projects

Type	Open-source, Rust-based, managed and self-hosted options
Performance	1ms p99 (small datasets), 626 QPS at 1M vectors, lower throughput >10M
Pricing	1GB free forever, $25/month paid, $99/month Hybrid Cloud

Choose when	Budget-conscious, need filtering, <50M vectors, edge deployments
Avoid when	>50M vectors, high concurrent writes, need largest ecosystem

If budget is a primary concern, **Qdrant** offers the best free tier in this comparison: 1GB of vector storage forever, no credit card required. For startups and developers evaluating options, that removes friction. You can build and test without spending money or worrying about trial expirations.

The value extends beyond the free tier. Paid plans start at $25 monthly, which is competitive. For teams on a budget, Qdrant delivers solid performance without the price tag of Pinecone or the operational complexity of Milvus. That’s the positioning: good performance, good features, good price. Not the absolute best at any single dimension, but strong across all three.

Filtering is where Qdrant stands out. The database supports **rich JSON-based filters** that integrate with vector search efficiently. You can filter by nested properties, combine multiple conditions, and the query planner optimizes execution. Metadata filtering is a common requirement in RAG systems (filter by document type, date range, user permissions), and Qdrant handles it well.

The Rust implementation keeps the footprint compact. Qdrant runs efficiently on smaller instances, which matters for edge deployments or on-device applications. You can deploy it on IoT devices, in mobile apps, or in resource-constrained environments where other databases would struggle. The performance per watt is good.

But there are limits. Benchmarks show performance degrading beyond 10 million vectors. At 50 million vectors, Qdrant achieves **41.47 QPS at 99% recall**, compared to pgvectorscale's 471 QPS. That's an order of magnitude difference. Teams with large datasets need to test Qdrant carefully at their expected scale before committing.

Concurrent writes can also be challenging. Some users report issues with high write volumes, though this depends on configuration and hardware. If your application updates vectors frequently under heavy load, verify that Qdrant meets your requirements. For read-heavy workloads or moderate write volumes, it handles well.

The ecosystem is smaller than Pinecone, Weaviate, or Milvus. LangChain and LlamaIndex both integrate with Qdrant, documentation is solid, but the community is smaller. You'll find fewer Stack Overflow answers, fewer blog posts, fewer examples. That's fine if you're comfortable reading official docs and working things out, but it's a factor.

Cloud deployment options include managed Qdrant Cloud, or you can deploy on Azure, GCP, and AWS using their Hybrid Cloud offering. The flexibility is there. Self-hosting is straightforward with Docker or Kubernetes.

You choose Qdrant when you're budget-conscious (best free tier, competitive paid plans), need complex filtering at moderate scale (under 50 million vectors), want edge or on-device deployment (compact footprint), or want a good balance of features without breaking the bank. You avoid it when you need to scale beyond 50 million vectors, when high concurrent writes are critical, or when you need the largest ecosystem and community support.

pgvector + pgvectorscale

Add vector search to your existing PostgreSQL infrastructure without managing a separate database

Type	PostgreSQL extensions, self-hosted or managed (cloud PostgreSQL)
Performance	471 QPS at 50M vectors (99% recall), sub-100ms, competitive with specialized DBs
Pricing	Free extensions, infrastructure costs only (your PostgreSQL server)
Choose when	Use PostgreSQL, want vectors+relational, <100M

	vectors, reduce complexity
Avoid when	>100M vectors, pure vector workload, no PostgreSQL expertise, greenfield optimization

The databases we've covered so far are purpose-built for vectors. But if you already run PostgreSQL, **pgvector** and **pgvectorscale** give you vector search without adding a new database. That's the core value: unified data management. Your vectors live next to your relational data, you query both in the same transaction, and you manage one system instead of two.

The performance case used to be weak. Earlier versions of `` `pgvector` `` were adequate for small datasets but couldn't compete with specialized vector databases. That changed with `` `pgvectorscale` `` from Timescale. Recent benchmarks show **471 QPS at 99% recall on 50 million vectors**, which is 11.4 times better than Qdrant and competitive with Pinecone's specialized infrastructure. The p95 latency is 28 times lower than Pinecone s1 at 99% recall.

The technical approach uses DiskANN with Statistical Binary Quantization, keeping vectors on disk efficiently while maintaining high recall. For teams already managing PostgreSQL at scale, adding vector search becomes an incremental step, not a new project. You use existing backup systems, monitoring tools, replication setup, and operational knowledge.

Cost savings are significant. Self-hosting `pgvector` on AWS costs about 75% less than Pinecone for comparable workloads. You're paying for PostgreSQL infrastructure you might already have. No additional licensing, no per-vector pricing, no query costs. Just compute and storage.

The limitations are real though. Beyond 100 million vectors, PostgreSQL's relational storage model hits walls that purpose-built vector databases avoid. Milvus and Pinecone are engineered for billions of vectors. PostgreSQL is a relational database with vector capabilities added on. At extreme scale, that architectural difference matters.

Concurrent vector queries are not optimized the way they are in specialized systems. If your workload is purely high-throughput vector search with thousands of queries per second, purpose-built databases will outperform. But if you're combining vector search with relational queries, transactions, and traditional database operations, the unified model is more efficient overall.

The learning curve is minimal if you know PostgreSQL. You install extensions, create indexes, and query using SQL. There's no new query language, no new operational patterns, no new mental models. Teams with PostgreSQL expertise can add vector search in days, not weeks. That deployment speed matters.

The ecosystem integration works through existing PostgreSQL tools. Any framework that supports PostgreSQL can use `` `pgvector` ``. ORMs, query builders, connection poolers, all work normally. You don't need special SDKs or libraries. SQL is the interface.

You choose `` `pgvector` `` and `` `pgvectorscale` `` when you already run PostgreSQL, need vectors alongside relational data, want to reduce system complexity (one database instead of two), operate at moderate scale (under 50 million vectors where it's highly competitive, up to 100 million where it still works), want significant cost savings, and have PostgreSQL expertise. You avoid it when you need to scale beyond 100 million vectors, when your workload is purely vector search at massive throughput, when you don't have PostgreSQL expertise, or when you're building greenfield and can choose the optimal architecture without legacy constraints.

Elasticsearch

Battle-tested search platform adding vector capabilities to proven full-text search and analytics

Type	Search engine with vector capabilities, self-hosted or managed
Performance	~260ms exact kNN, 5x faster than OpenSearch, reliable at 50M+ vectors
Pricing	Elastic Cloud or self-hosted with infrastructure costs
Choose when	Already use Elasticsearch, need hybrid search, value operational maturity
Avoid when	Pure vector workload, need lowest latency, greenfield, very cost-sensitive

Like pgvector extends PostgreSQL, Elasticsearch is a search engine first, vector database second. That ordering matters. If you already run Elasticsearch for full-text search, logging, or analytics, adding vector capabilities makes sense. You use existing infrastructure, operational knowledge, and tooling. If you don't run Elasticsearch, starting with it just for vectors is probably the wrong move.

The reliability is proven. Elasticsearch has been running at massive scale in production for over a decade. The operational patterns are well-understood, the failure modes are documented, the monitoring tools are mature. When you add vector search to an existing Elasticsearch cluster, you inherit that stability. For enterprise teams that value operational maturity over the newest performance features, that's a strong argument.

Vector performance has improved a lot. **Elasticsearch 8.14 is 5 times faster than OpenSearch** for vector search using Binary Quantized Vectors, with 75% cost reduction and 50% faster indexing. The HNSW implementation with 8-bit and 4-bit quantization delivers sub-50ms kNN queries even with term and range constraints. That's competitive for many use cases.

But specialized vector databases are still faster. Compare Elasticsearch's ~260ms for exact kNN to Pinecone's 7ms p99 or Milvus's low single-digit millisecond performance. The gap narrows with approximate search and quantization, but it's still there. If you need absolute minimum latency for pure vector workloads, purpose-built databases win.

Hybrid search is another strength. Elasticsearch combines BM25 (a ranking algorithm for full-text search) with vector similarity using **Reciprocal Rank Fusion**. You can query by keywords, apply filters, add vector similarity, and get unified results. For applications that need traditional search and semantic search together, this integration is clean. Weaviate does this too, but if you already have Elasticsearch, you don't need to add another system.

The cost model depends on deployment. Elastic Cloud pricing is complex and can get expensive at scale. Self-hosting saves money but requires managing

Elasticsearch clusters, which is non-trivial. Elasticsearch is licensed under dual SSPL and Elastic License (changed from Apache 2.0 in 2021), so understand the terms if you're self-hosting commercially.

Resource requirements are higher than specialized vector databases for the same workload. Elasticsearch uses more memory and compute because it's doing more than just vector search. You're running a full search engine with aggregations, analytics, and other features. That overhead is fine if you use those features. It's wasteful if you only need vectors.

The learning curve exists. Elasticsearch has its own query DSL, its own concepts (indices, shards, segments), its own operational patterns. Teams already familiar with it move quickly. New users face a steeper ramp than they would with Pinecone or Qdrant. The documentation is extensive but also overwhelming for beginners.

For teams already running Elasticsearch, adding vector capabilities leverages existing infrastructure and operational knowledge. The decade of production hardening and mature tooling provide reliability that newer vector databases can't match. Mixed workloads (logs, metrics, traditional search, vectors) benefit from the unified platform. Starting with Elasticsearch purely for vector search makes less sense—purpose-built databases offer better latency and simpler deployment for greenfield projects.

Specialized Database Options

The top six vector databases cover most use cases, but some scenarios need specialized tools. These eight databases excel in specific scenarios. If you have specialized requirements beyond general RAG and semantic search, evaluate these alongside the top 6.

ChromaDB

The fastest path from idea to prototype with a developer-friendly API and embedded architecture

Best for	Prototyping, learning, MVPs under 10M vectors
Why it's special	Best developer experience, embedded architecture, NumPy-like API, built-in features
Limitation	Not as fast as specialized databases, not designed for production scale
Pricing	Free (Apache 2.0), Chroma Cloud offers $5 free credits

ChromaDB gives you the fastest path from idea to working prototype. The

Python API feels like NumPy, not a database. You initialize it, add embeddings, query. No configuration, no setup, no infrastructure. It runs embedded in your application with zero network latency. That's why developers choose it for learning vector databases and building MVPs.

The 2025 Rust rewrite delivers **4x faster writes and queries** compared to the original Python implementation. It's not as fast as specialized databases like Qdrant or Pinecone. But for prototypes under 10 million vectors, that performance difference doesn't matter. Getting your RAG system working quickly matters more.

Built-in metadata and full-text search mean you don't need to integrate separate tools. ChromaDB handles filtering and keyword search alongside vector similarity. For a prototype, this is one less thing to build. The Apache 2.0 license means it's completely free for any use.

The limitation is scale. ChromaDB isn't designed for production workloads at 50 million or 100 million vectors. It's designed for development speed, not operational scale. Teams outgrow it and migrate to Qdrant, Pinecone, or Milvus when they go to production.

Ideal for rapid prototyping, learning, and MVPs under 10 million vectors. The NumPy-like Python API and embedded architecture remove deployment friction entirely. For a real-world example of

ChromaDB in action, see this **code documentation RAG assistant** that demonstrates practical implementation. Teams transition to production-grade databases (Qdrant, Pinecone, Milvus) as they scale, but ChromaDB excels at getting ideas working fast.

Redis Vector Search

Ultra-low latency vector search leveraging your existing Redis cache for sub-millisecond performance

Best for	Lowest latency requirements under 10M vectors
Why it's special	Sub-ms latency, 62% more throughput, 66K-160K insertions/sec, 75% memory reduction with quantization
Limitation	Memory-bound, expensive beyond 10M vectors
Pricing	Redis Cloud pricing varies, memory costs dominate

For applications requiring ultra-low latency, **Redis** already sits in your infrastructure for caching. Adding vector search to it removes the need for another database. That's the value: if you need sub-millisecond latency and already use Redis, the vector extension makes sense.

In-memory architecture delivers the lowest query latency in this comparison. Benchmarks show **62% more throughput than competitors** for lower-dimensional datasets. Write performance reaches 66,000 to 160,000 vector insertions per second at billion scale, depending on precision configuration. The int8 quantization provides 75% memory reduction while maintaining 99.99% accuracy.

But memory is expensive. Storing 10 million vectors of 1536 dimensions requires about 60GB of RAM. At cloud memory pricing, that costs more than disk-based alternatives. Beyond 10 million vectors, the cost advantage shifts to databases like Milvus or Pinecone that use disk storage with smart caching.

Applications requiring sub-millisecond latency at moderate scale (under 10 million vectors) fit Redis well, especially if already deployed for caching. Beyond that threshold, memory costs make disk-based alternatives more economical.

FAISS

Meta’s proven similarity search library for custom billion-scale implementations with GPU acceleration

Best for	Billion-scale similarity search with custom integration
Why it’s special	Meta-proven at billions, 8.5x faster than previous methods, GPU-accelerated, memory-efficient
Limitation	Library only, requires significant integration, no built-in

	persistence/APIs
Pricing	Free (open-source library)

Unlike the databases we've covered, FAISS from Meta Research is a similarity search library, not a complete database. That distinction matters. It doesn't include persistence, APIs, or management features. You integrate it into your application and build those layers yourself. The trade-off is control and performance at massive scale.

Meta uses FAISS in production at billions of vectors. The library is **8.5x faster than previous best methods** for similarity search. GPU acceleration handles millions of similarity checks in milliseconds. Memory-efficient compression techniques let you store massive datasets economically. This is proven technology at extreme scale.

The trade-off for this performance is integration work. You write the persistence layer, build the APIs, create management tools, implement monitoring. For teams with engineering resources and specific requirements that off-the-shelf databases don't meet, this investment makes sense. For most teams, using a complete database is more practical.

You choose FAISS when you need billions of vectors, have engineering resources for custom integration, require GPU acceleration, or want maximum control over the similarity search implementation.

You avoid it when you need a complete database solution, want minimal integration work, or don't have specialized requirements that justify the development effort.

Marqo

Purpose-built for multi-modal search across text, images, audio, and video with a unified API

Best for	Multi-modal AI (text + images + audio + video)
Why it's special	Built for multi-modal, ecommerce models outperform Amazon Titan by 88%, single API for all media
Limitation	Newer/less proven, smaller ecosystem, limited benchmarks
Pricing	Check official website (pricing structure emerging)

Moving from low-level libraries to specialized use cases, **Marqo** specializes in multi-modal AI: text, images, audio, and video in one search system. If your application searches across different media types, Marqo handles that directly. Their proprietary ecommerce models **outperform Amazon Titan Multimodal Embedding by up to 88%** for product search, with a 94.5% increase in NDCG@10 for in-domain evaluations.

The end-to-end approach means you send raw content (images, text, video) to a single API. Marqo handles embedding generation, storage, and retrieval. You don't integrate separate embedding models or coordinate different systems. For multi-modal applications, this simplifies the architecture significantly.

The limitation is maturity. Marqo is newer than established players like Pinecone or Milvus. The ecosystem is smaller, benchmark data is limited, and production case studies are fewer. Early adopters report good results, but it hasn't been proven at the scale of other options.

Multi-modal applications (text, images, audio, video) benefit from Marqo's end-to-end approach, particularly for ecommerce search. The smaller ecosystem and limited production case studies reflect its newer status. Text-only applications have more mature alternatives.

DataStax Astra DB

Real-time vector updates with immediate consistency for operational workloads on Cassandra

Best for	Real-time vector updates + operational data
Why it's special	Immediate vector update availability, 6-9x faster than Pinecone (GigaOm), Cassandra-based, multi-model
Limitation	Requires Cassandra expertise, enterprise pricing

Pricing	Enterprise (contact sales), Azure/GCP marketplace

For real-time operational workloads, **DataStax Astra DB** builds on Cassandra, which means it's designed for distributed workloads and operational data. The unique claim is immediate vector update availability. When you update a vector, it's immediately queryable. Other databases have eventual consistency windows where updates propagate. For real-time applications that frequently modify vectors, this matters.

Performance is competitive, with **GigaOm testing showing 6-9x better performance than Pinecone** in RAG workloads. The multi-model approach handles tabular data, search, graph, and vectors in one system. For applications with mixed workload types, especially streaming data, this unified platform reduces complexity.

But there's a trade-off in Cassandra expertise. If your team doesn't understand Cassandra's architecture, consistency models, and operational patterns, there's a learning curve. The pricing is enterprise-focused (contact sales), which signals the target market: larger organizations with complex requirements, not startups or individual developers.

Consider Astra DB when real-time vector updates with immediate consistency matter—streaming data architectures and operational workloads that modify vectors frequently under load. Cassandra expertise and enterprise budgets are prerequisites. Simple RAG applications with infrequent updates don't need this level of consistency.

MongoDB Atlas Vector Search

Native vector search within MongoDB for unified document and embedding queries

Best for	MongoDB users needing vectors + document data
Why it's special	Vectors within MongoDB, sub-50ms at 15.3M vectors (with quantization), hybrid queries, unified data
Limitation	Document database first, not optimized for pure vector at massive scale
Pricing	$5K-70K/year (integrated into Atlas)

Similarly, if you're already using MongoDB, **MongoDB Atlas Vector Search** puts vectors inside MongoDB, next to your document data. If you already store product catalogs, user profiles, or content in MongoDB, adding vector search doesn't require syncing between databases. You query vectors and documents together, combine vector similarity with aggregation pipelines, and manage one system.

The implementation uses HNSW indexing with scalar and binary quantization. Performance shows **sub-50ms query latency at 15.3 million vectors (with quantization, 90-95% accuracy)**. It supports up to 4096 dimensions, which covers most embedding models. Hybrid queries work naturally because vector search integrates with MongoDB's existing query language.

The limitation is focus. MongoDB is a document database first, vector database second. At massive scale with pure vector workloads, specialized databases like Milvus or Pinecone will outperform. But if you need vectors alongside document operations, the unified model is more efficient than managing separate systems.

Pricing integrates into MongoDB Atlas, ranging from $5,000 to $70,000 annually depending on scale and features. That's enterprise pricing. For teams already paying for Atlas, adding vectors is incremental. For teams not using MongoDB, it's a significant commitment.

You choose MongoDB Atlas Vector Search when you already use MongoDB, need vectors and documents in the same queries, want unified data management, or run applications where document operations and vector search happen together. You avoid it when you need a pure vector database optimized for maximum performance, don't use MongoDB, or want dedicated vector-first architecture.

Zilliz Cloud

Enterprise-managed Milvus with 10x performance boost and billion-scale capabilities

Best for	Enterprise-scale (100M-1B+ vectors) managed Milvus
Why it's special	10x faster than OSS Milvus, sub-10ms p50, scales to 100B items, 70% lower TCO
Limitation	More expensive than self-hosted Milvus, steeper learning curve than Pinecone
Pricing	Free tier (5GB), $99/month dedicated, $4/M vCUs serverless

We covered Milvus earlier as a self-hosted option. **Zilliz Cloud** is managed Milvus with enterprise features. The Cardinal engine delivers **10x faster vector retrieval than open-source Milvus**, with sub-10ms p50 latency. It scales to 500 compute units and over 100 billion items. The 70% lower total cost of ownership claim comes from better resource utilization and automated optimization.

This is the choice when you want Milvus architecture and performance without managing it yourself. You get 99.95% uptime SLA, SOC2 Type II and ISO27001 certification, and enterprise support. The operational burden of running Milvus disappears, but you still benefit from its billion-scale capabilities.

The trade-off compared to self-hosted Milvus is cost. Managed services always cost more than running infrastructure yourself. The free tier provides 5GB storage with 2.5 million vCU limits, which is enough for testing. Paid plans start at $99 monthly for dedicated clusters, or $4 per million vCUs on serverless.

Learning curve is similar to Milvus. You still need to understand vector indexing concepts, HNSW parameters, and Milvus architecture. It's easier than Pinecone for operational simplicity but not as simple. The target market is teams that need Milvus scale and have data engineering talent, but want to focus on their application instead of database operations.

Best fit for teams needing Milvus architecture and billion-scale capabilities without self-hosting complexity. The 99.95% SLA and SOC2/ISO compliance suit enterprise requirements. Teams wanting the simplest managed experience should choose Pinecone; those comfortable self-hosting can run Milvus directly for lower costs.

SingleStore

Unified platform combining real-time analytics and vector search for complex workloads

Best for	Real-time analytics + vector search unified

Why it's special	Analytics+vectors combined, 1.6-6x faster indexing than Milvus, streaming, multi-model
Limitation	Higher complexity, opaque pricing, learning curve
Pricing	Enterprise (contact sales)

Finally, SingleStore combines real-time analytics with vector search. The architecture handles analytical queries (aggregations, joins, time-series) alongside vector similarity in one system. This matters for applications that need both: analyzing user behavior while also serving personalized recommendations based on vector embeddings.

Vector performance is competitive with specialized databases. SingleStore uses HNSW and IVF indexes comparable to Milvus, with 1.6 to 6 times faster index building than Milvus in their benchmarks. It supports vectors plus JSON, time-series, full-text search, spatial data, and key-value operations. The SQL interface means you can combine all these in single queries.

Streaming data ingestion is built-in. If you're processing real-time data streams and need vector search on that data, SingleStore handles both without separate systems. Apache Iceberg integration (added in 2025) connects to data lakes.

The complexity is higher than pure vector databases. SingleStore is a full analytical database with many features. Learning the system takes time. Pricing is enterprise-focused and not transparent (contact sales), which makes it harder to evaluate for smaller teams or projects.

Real-time analytics combined with vector search in a unified system distinguishes SingleStore. Applications analyzing user behavior while serving personalized recommendations based on embeddings benefit from the integrated approach. Pure vector search workloads don't need the analytical complexity or enterprise pricing model.

Building a RAG Pipeline: From Web to Vector Database

You've picked a vector database. Now you need data to put in it.

A typical RAG workflow follows these steps: collect data, clean it, chunk it, embed it, and store it. In our situation, the main challenge is the first step—we don't have any data yet. For most AI projects, the first and most straightforward action is to gather data from online sources.

Suppose you want to compare how different vector databases perform with medical data. The first requirement is to obtain real medical content from the web. However, standard web scraping methods often return HTML filled with navigation menus, JavaScript, ads, and other unwanted elements, rather than the main content you need.

Firecrawl solves this problem by extracting clean content from web pages and converting it to markdown or structured JSON. You simply provide a URL, and Firecrawl manages JavaScript rendering and removes boilerplate, giving you clean text that is ready for embedding.

The workflow

Here's what a complete pipeline looks like:

1. Scrape web content with Firecrawl
2. Save the clean markdown
3. Load and chunk the documents
4. Generate embeddings
5. Store in your vector database
6. Query

Let's build this pipeline with Pinecone using code, then you'll see how to swap in other databases. If you prefer visual workflow builders, check out this **LangFlow tutorial** for building similar pipelines without code.

Step 1: Collecting data with Firecrawl

First, install Firecrawl and get an API key from **firecrawl.dev**:

```
uv pip install firecrawl-py python-
```

Save your API key to a `.env` file:

```
FIRECRAWL_API_KEY=fc-YOUR-KEY-HERE
```

The script below crawls a website to discover pages, then scrapes each page and saves the content as markdown. Let’s break it down step by step.

Initialize Firecrawl and discover pages

```
from firecrawl import Firecrawl
from dotenv import load_dotenv
from pathlib import Path

load_dotenv()
app = Firecrawl()

# Crawl to discover pages
crawl_result = app.crawl(
    "https://www.mayoclinic.org/dr
    limit=10,
    scrape_options={'formats': ['m
)
```

This initializes the Firecrawl client and crawls the Mayo Clinic drugs page to discover up to 10 related pages, requesting markdown format for easy processing. For more advanced crawling capabilities and configurations, see the guide on mastering Firecrawl's crawl endpoint. Once we have those pages, we can scrape them all at once.

Extract URLs and batch scrape

```
# Extract URLs
urls = [page.metadata.url for page
        if page.metadata and page.m

# Batch scrape
batch_job = app.batch_scrape(urls,
```

Here we extract the discovered URLs from the crawl results, filtering out any pages without valid URLs, then batch scrape all pages in parallel to get their content as markdown. Now we need to save this content for later processing.

Save markdown files

```
# Save as markdown files
output_dir = Path("data/documents")
output_dir.mkdir(parents=True, exis

for i, result in enumerate(batch_jo
    filename = f"doc_{i:02d}.md"
    with open(output_dir / filename
        f.write(result.markdown)
```

This creates a directory for our documents and saves each scraped page as a numbered markdown file. You now have clean markdown files ready for processing, without any HTML or boilerplate content.

Output:

```
Discovered 10 URLs
Saved 10 documents to data/document
```

With your data collected and cleaned, you can move on to building the actual RAG system.

Step 2: Building the RAG application with Pinecone

We'll load those markdown files, chunk them, and store them in Pinecone. First, install the required packages:

```
uv pip install langchain-pinecone l
```

Set up your environment variables for OpenAI embeddings integration:

```
PINECONE_API_KEY=your-key
OPENAI_API_KEY=your-key
```

The code below builds the complete RAG pipeline. We'll walk through each step.

Import dependencies and load environment

```
import os
from pathlib import Path
from dotenv import load_dotenv
from pinecone import Pinecone, Serv
from langchain_openai import OpenAI
from langchain_pinecone import Pine
from langchain_text_splitters impor
from langchain_core.documents impor

load_dotenv()
```

This imports all necessary libraries for vector storage (Pinecone), embeddings (OpenAI), document processing (LangChain), and environment configuration. With these in place, we can set up the Pinecone index.

Initialize Pinecone and create index

```
# Initialize Pinecone
pc = Pinecone(api_key=os.environ["
index_name = "drug-info-rag"

# Create index if needed
if not pc.has_index(index_name):
    pc.create_index(
        name=index_name,
        dimension=1536,
        metric="cosine",
        spec=ServerlessSpec(cloud=
    )
```

This connects to Pinecone and creates a serverless index configured for OpenAI's `text-embedding-3-small` model (1536 dimensions) using cosine similarity for semantic search. Now we need to connect our embedding model to this index.

Set up embeddings and vector store

```
# Set up vector store
embeddings = OpenAIEmbeddings(model
index = pc.Index(index_name)
vector_store = PineconeVectorStore(
```

This initializes the OpenAI embeddings model and connects it to our Pinecone index through LangChain's vector store wrapper, which handles embedding generation and storage automatically. Time to load the documents we scraped earlier.

Load documents

```
# Load documents
documents = []
for md_file in Path("data/documents
    with open(md_file) as f:
        documents.append(
            Document(page_content=f
        )
```

This reads all markdown files from the documents directory and converts them to LangChain `Document` objects, preserving the source filename as metadata for later reference. Before we can embed these documents, we need to split them into smaller chunks.

Chunk documents

```
# Chunk documents
splitter = RecursiveCharacterTextSp
    chunk_size=500,
    chunk_overlap=50
)
```

```
chunks = splitter.split_documents(d
```

This splits documents into 500-character chunks with 50-character overlap to maintain context across boundaries. The recursive splitter intelligently breaks at natural boundaries like paragraphs and sentences. With our chunks ready, we can upload them to Pinecone and test a query.

Add to Pinecone and query

```
# Add to Pinecone
vector_store.add_documents(chunks)

# Query
results = vector_store.similarity_
    "What are the side effects of
    k=3
)

for result in results:
    print(f"From {result.metadata[
    print(result.page_content[:200
```

This uploads all chunks to Pinecone with their embeddings, then performs a semantic search for relevant information about sertraline side effects, retrieving the top 3 most similar chunks.

Output:

```
Query: "What are the side effects c

From Sertraline_oral_route_-_Side_e
Common side effects of sertraline m

From doc_01.md:
Sertraline can cause side effects s

From doc_01.md:
 ### Geriatric  Studies show no uniq
```

You now have a working RAG system that can answer questions based on web content. The best part? You can swap out Pinecone for a different vector database with minimal code changes.

Adapting for other vector databases

The pattern stays the same across all databases: collect data with Firecrawl, chunk it, embed it, store it. Only the vector store setup changes.

For **Qdrant**:

```
from langchain_qdrant import Qdrant
from qdrant_client import QdrantCli

client = QdrantClient(url="http://l
vector_store = QdrantVectorStore(
    client=client,
    collection_name="drug-info",
    embedding=embeddings
)
vector_store.add_documents(chunks)
```

For **Weaviate**:

```
from langchain_weaviate import Weav
import weaviate

with weaviate.connect_to_local() as
    vector_store = WeaviateVectorSt
        client=client,
        index_name="DrugInfo",
        text_key="text",
        embedding=embeddings
    )
    vector_store.add_documents(chun
```

For **ChromaDB**:

```
from langchain_chroma import Chroma

vector_store = Chroma(
    collection_name="drug-info",
    embedding_function=embeddings,
    persist_directory="./chroma_db"
)
```

```
vector_store.add_documents(chunks)
```

For **pgvector** (PostgreSQL):

```
from langchain_postgres import PGVe

connection_string = "postgresql://u
vector_store = PGVector(
    connection_string=connection_st
    collection_name="drug_info",
    embedding_function=embeddings
)
vector_store.add_documents(chunks)
```

The data collection part stays identical across all of these. Firecrawl gives you clean content, you chunk it, and the vector store handles the rest. Switch databases by changing about 10 lines of code—the entire workflow from scraping to querying remains the same.

Conclusion

There's no single best vector database. The right choice depends on your scale, infrastructure, and budget. Pinecone excels at ease of use, Milvus at cost efficiency for billions of vectors, pgvector at PostgreSQL integration, and Weaviate at hybrid search. The **decision framework** maps these to specific use cases.

Those guidelines give you a starting point, but the real test is your own data. Your document types and embedding models will shift the performance numbers. Build a small test with 1,000 documents (**Building a RAG Pipeline** covers extracting clean data from websites with Firecrawl), run your actual query patterns, and measure recall and latency. Pick what works for your use case, not what benchmarks say should work.

// Get started \\

Ready to build?

Start getting Web Data for free and scale seamlessly as your project expands. No credit card needed.

Start for free See our plans

Bex Tuychiev @bextuychiev
Technical Writer at Firecrawl

About the Author

Bex Tuychiev is a Technical Writer at Firecrawl and a Kaggle Master with over 15k followers. He loves writing detailed guides, tutorials, and notebooks on complex data science and machine learning topics

More articles by Bex Tuychiev

Data Enrichment: A Complete Guide to Enhancing Your Data Quality

Top 10 Browser Automation Tools for Web Testing and Scraping in 2026

How to Create a Claude Code Skill: A Web Scraping Example with Firecrawl

Best Open-Source Web Crawlers in 2026

Scraper vs Crawler: When to Use Each (With Examples)

15 Best Open-Source RAG Frameworks in 2026

Python Web Scraping Tutorial - Setup & Examples

Best Open-Source Web Scraping Libraries in 2026

8 Best Web Scraping APIs in 2025

How to Scrape Dynamic Websites with Headless Browsers in Python

FOOTER

Firecrawl

The easiest way to extract data from the web

Backed by

Y Combinator

SOC II · Type 2

AICPA
SOC 2

Linkedin

X (Twitter)

Github

Discord

YouTube

Products

Use Cases

Playground	AI Platforms
Extract	Lead Enrichment
Pricing	SEO Teams
Templates	Deep Research
Changelog	Competitive Intelligence

Documentation	Company
Getting started	Blog
API Reference	Careers
Integrations	Creator & OSS program
Examples	Student program
SDKs	

© 2025 Firecrawl

Terms of Service

Privacy Policy

Report Abuse

All systems normal