Architectural Blueprint for Scalable AI-Driven Fintech Systems: Reaching One Million Users with Python-Centric Infrastructure

1. Executive Architecture and Scale Analysis

The trajectory of modern financial technology is defined by a rigorous demand for hyper-scale availability, absolute data consistency, and increasingly, the integration of autonomous agentic intelligence. The transition from a proof-of-concept application to a platform supporting one million active users represents a fundamental phase shift in engineering requirements. At this magnitude, the system is no longer merely a web application; it becomes a distributed financial utility where the cost of latency or data inconsistency is existential.

This report provides an exhaustive analysis of the architectural requirements for the "SCALE" financial analytics application as it evolves from a Next.js/Supabase prototype into a robust, AI-driven financial platform. The analysis is grounded in the specific codebase context provided—a dashboard-centric application aiming to replace human accountants with AI agents—and expands into a comprehensive blueprint for handling high-frequency transaction data, predictive modeling, and regulatory compliance at scale.

1.1 The "One Million User" Constraint in Fintech

Defining "one million users" in a fintech context requires a nuanced understanding of load profiles. Unlike social media, where read operations vastly outnumber writes, a financial "AI Accountant" generates a write-heavy workload. Every transaction, every market tick, and every user interaction triggers a cascade of downstream effects: ledger updates, anomaly detection inferences, budget recalculations, and forecasting adjustments.

If we assume a conservative estimate where 1 million users generate an average of 5 transactions per day (including card swipes, bank transfers, and automated bill payments), the system must handle 5 million core transaction writes daily. However, the *derived* load is significantly higher. Each transaction might trigger:

1. **Vector Embedding:** Converting the merchant string to a vector for categorization.
2. **Inference Requests:** Querying the anomaly detection model (Isolation Forest) and the spending forecast model (TFT).
3. **Notification Events:** Calculating if a budget threshold has been breached.
4. **Read Queries:** The user checking their dashboard immediately after a purchase.

This implies a peak load that can easily exceed 50,000 requests per second (RPS) during high-volume windows (e.g., paydays or holidays). A monolithic architecture, where a single database instance handles both the transactional writes (OLTP) and the analytical queries (OLAP) required by the AI, will inevitably suffer from resource contention. The database locks required to ensure ACID compliance for the ledger will block the long-running scans required by the prediction engines.[1]

1.2 Analysis of the Current SCALE Repository State

The current state of the SCALE project, as inferred from the provided development logs, rests on a foundational stack of Next.js for the frontend and Supabase (PostgreSQL) for the backend. The application currently features:

- **Overview Page (Dashboard):** Likely consuming aggregated views of transaction data.
- **Transactions Page:** A granular view of the ledger.
- **Analytics Page:** Visualization components (charts, heatmaps, leaderboards).
- **Authentication:** Multi-account Google OAuth.
- **Privacy Layer:** A privacy.ts file indicating an intent to handle data anonymization on the client side or at the edge.[3]

While this stack is excellent for velocity and rapid prototyping, it presents specific bottlenecks for the target scale:

1. **Next.js API Routes:** Running heavy AI logic (like Python-based ML inference) within Node.js serverless functions (Vercel) is inefficient. It introduces "cold start" latency and lacks the specialized libraries available in the Python ecosystem (PyTorch, Pandas, Scikit-Learn).
2. **Supabase as a Monolith:** While Supabase scales vertically well, using a single Postgres instance for both the live ledger and the training data for ML models creates a single point of failure. Complex analytical queries for "Cash Flow Forecasting" could degrade the performance of the "Transaction Insert" operations.
3. **Data Ingestion:** The current "Smart Import" relies on file uploads or real-time syncing. At scale, relying on client-side parsing or synchronous API calls for bulk imports (e.g., a user uploading 5 years of bank statements) will time out standard HTTP connections.

1.3 The Microservices Migration Strategy

To support the vision of an "AI Accountant," the architecture must evolve into a set of specialized microservices. The core philosophy here is "Decoupled Intelligence." The logic that manages the *money* (Ledger Service) must be physically and logically separated from the logic that *thinks* about the money (AI Prediction Service).

We propose a migration to a **Python-Centric Backend** for the intelligence layer, specifically leveraging **FastAPI** for high-concurrency orchestration and **Ray Serve** for distributed model inference. The Next.js layer remains as the "Presentation Tier," responsible for rendering the

UI and managing the WebSocket/SSE connections for real-time feedback, but it delegates all heavy lifting to the Python backend.[4]

Component	Current State	Target State (1M Users)	Rationale
Frontend	Next.js (Monolith)	Next.js (BFF - Backend for Frontend)	Decouples UI from logic; optimize for Edge rendering.
API Layer	Next.js API Routes	FastAPI Gateway	Async Python support; superior ecosystem for AI integration.
Compute	Serverless Functions	Ray Serve Cluster	Distributed inference; fractional GPU scaling; stateful agents.
Database	Supabase (Postgres)	Polyglot (Citus + ClickHouse + Qdrant)	Separation of concerns: OLTP vs. OLAP vs. Vector Search.
Communication	REST (Synchronous)	Event-Driven (Kafka/Redpanda)	Decoupling writes from processing; fault tolerance.

This architectural shift is not merely about changing languages; it is about changing the *flow of data*. In the new architecture, a transaction is not "processed" when it is received. It is "accepted" and placed onto an event bus. The processing happens asynchronously, allowing the system to absorb massive spikes in traffic without degrading the user experience.

2. The Data Layer: Polyglot Persistence and Regulatory Compliance

Data is the lifeblood of any fintech application, but for an AI-driven platform, it is also the fuel for intelligence. The storage architecture must satisfy three competing objectives:

Consistency (the balance must be correct), **Latency** (the dashboard must load instantly), and **Privacy** (the data must be secure).

2.1 The Account Aggregator (AA) Framework Integration

For the Indian market, and increasingly globally via Open Banking standards, the ingestion of financial data is standardized through the Account Aggregator (AA) framework. This is a critical component of the "Smart Import" engine mentioned in the project scope. The AA framework acts as a blind data pipe, ensuring that data flows from Financial Information Providers (FIPs) to Financial Information Users (FIUs) with explicit user consent and end-to-end encryption.

2.1.1 The ReBIT API Workflow

Implementing the AA integration requires strict adherence to the ReBIT (Reserve Bank Information Technology) API specifications. The workflow for the SCALE app (acting as an FIU) involves a complex sequence of cryptographic handshakes:

1. **Consent Initiation:** The SCALE app generates a **Consent Request** payload. This is not a simple permission prompt; it is a structured XML/JSON artifact defined by the ReBIT schema. It specifies the *Data Life* (how long the data is stored), *Data Frequency* (fetch interval), and *Purpose Code* (e.g., "101 - Wealth Management").
2. **Consent Artefact:** Once the user approves the request via their AA app (e.g., Anumati, OneMoney), the AA generates a digitally signed **Consent Artefact**. This artefact is a cryptographic proof of permission.
3. **FI Request (Data Fetch):** The SCALE app presents this artefact to the FIP (the bank) to request data. This request includes a **Diffie-Hellman Public Key** generated ephemerally for this specific session.
4. **End-to-End Encryption:** The FIP validates the artefact. If valid, it generates its own ephemeral key pair. It computes a shared secret using its private key and the SCALE app's public key (using ECDH - Elliptic Curve Diffie-Hellman). The financial data (bank statements) is encrypted using this shared secret.
5. **Data Delivery:** The encrypted payload flows through the AA to the SCALE app. The AA *cannot* decrypt this data because it does not possess the private keys of either party.
6. **Decryption:** The SCALE app receives the payload, uses its private key and the FIP's public key to reconstruct the shared secret, and decrypts the data locally.[6]

This architecture ensures that the SCALE app builds a "Proprietary Moat" not just in data volume, but in *data trust*. By implementing the AA framework natively, the app guarantees that it never relies on screen-scraping (which is fragile and insecure) and maintains a regulatory-compliant audit trail of all data access.[9]

2.2 Database Partitioning and Sharding Strategies

Managing the ledger for one million users requires moving beyond a single monolithic

PostgreSQL instance. We employ a strategy of **Horizontal Sharding**.

2.2.1 Transactional Sharding with Citus

Using **Citus**, an extension that transforms Postgres into a distributed database, we can shard the transactions table across multiple physical nodes. The choice of the **Shard Key** is critical. We select user_id as the distribution column.

- **Colocation:** All data related to a specific user (transactions, accounts, budgets, tags) is stored on the same physical worker node.
- **Performance:** When the "AI Accountant" queries a user's history to generate a forecast, the query is routed to a single node. This avoids "scatter-gather" queries that hit every node in the cluster, drastically reducing latency.
- **Scalability:** As the user base grows from 1 million to 10 million, we can simply add more worker nodes and rebalance the shards without downtime.[11]

2.2.2 Analytical Storage with ClickHouse

While Postgres/Citus handles the transactional ledger (the "truth"), it is suboptimal for the heavy aggregations required by the "Cash Flow Engine" (e.g., "Average spending on 'Food' on 'Fridays' over the last 5 years"). For this, we utilize **ClickHouse**, a columnar OLAP database.

- **Data replication:** An asynchronous CDC (Change Data Capture) pipeline (using Debezium and Kafka) replicates data from Postgres to ClickHouse.
- **Compression:** ClickHouse's columnar storage allows for massive compression ratios (often 10:1), enabling the cost-effective storage of years of transaction history.
- **Speed:** Aggregation queries that take seconds in Postgres take milliseconds in ClickHouse, enabling real-time analytics on the dashboard.[13]

2.3 Vector Database for Semantic Search

To power the "AI Accountant's" ability to answer natural language queries ("How much did I spend on that trip to Paris?"), we require a **Vector Database**.

- **Technology Selection:** We compare **Milvus** and **Qdrant**. For this architecture, **Qdrant** is selected due to its superior support for **HNSW (Hierarchical Navigable Small World)** indexing combined with **payload filtering**.
- **Implementation:** When a transaction arrives, an embedding model (e.g., all-MiniLM-L6-v2 or a fine-tuned financial BERT) converts the transaction narrative ("STARBUCKS STORE 0324") into a vector. This vector is stored in Qdrant along with the user_id payload.
- **Querying:** When a user asks a question, the system filters the vector search by user_id *before* traversing the HNSW graph. This ensures strict tenant isolation and prevents the "noisy neighbor" problem where one power user degrades search performance for others.[14]

3. The AI Engine Core: Proprietary Models and Mathematics

To achieve the goal of "replacing accountants," the SCALE app must transcend simple rule-based logic. It requires deep learning models capable of understanding the stochastic and complex nature of human financial behavior. We detail the architecture of the three primary engines identified in the project scope.

3.1 Forecasting Engine: Temporal Fusion Transformers (TFT)

Predicting future spending is a **Multi-Horizon Time Series Forecasting** problem. Traditional models like ARIMA or even simple LSTMs struggle because they cannot effectively integrate heterogeneous data types (static metadata vs. time-varying inputs).

We recommend implementing the **Temporal Fusion Transformer (TFT)** architecture. This is a significant upgrade over the XGBoost baseline initially considered in the logs.

Mathematical Architecture of TFT:

1. **Gating Mechanisms (GRN):** TFT employs Gated Residual Networks to filter out irrelevant inputs. For a user with a fixed salary, the "Day of Week" might be irrelevant for income prediction, while for a gig economy worker, it is critical. The GRN learns these weights per user.

   $$GRN_\omega(a, c) = LayerNorm(a + GLU_\omega(\eta_1))$$

   Where GLU is the Gated Linear Unit that controls the flow of information.
2. **Variable Selection Networks:** Unlike standard transformers that treat all inputs as a single vector, TFT employs variable selection networks to weight the contribution of each input variable independently at each time step. This allows the model to "explain" that a spike in predicted spending is specifically due to the "Holiday" variable.
3. **Temporal Processing:**
   - **Static Covariate Encoders:** Encode user profile data (Age, Income Bracket).
   - **LSTM Layers:** Process local temporal patterns (recent spending trends).
   - **Multi-Head Attention:** captures long-term dependencies (e.g., an annual insurance payment). The attention mechanism allows the model to "look back" 12 months and attend to the previous recurrence of a bill.

Why this wins: This architecture provides **Interpretability**. The "AI Accountant" can tell the user: "I predict a low balance next week *because* (Attention Weight High) of your recurring rent payment and *because* (Variable Selection) it is a holiday weekend." This builds trust, which is the currency of fintech.[17]

3.2 Categorization Engine: Hyperbolic Category Discovery (HypCD)

Transaction categorization is traditionally treated as a classification problem in Euclidean space. However, merchant categories are hierarchical (e.g., *Expense -> Food -> Restaurant -> Sushi*). Embedding hierarchies in Euclidean space causes distortion because the volume of Euclidean space grows polynomially, while the number of nodes in a hierarchy grows exponentially.

We propose a proprietary implementation of **Hyperbolic Category Discovery (HypCD)**.

- **The Poincaré Ball Model:** We embed transaction strings into a Hyperbolic space (specifically the Poincaré ball). In this geometry, the volume grows exponentially with the radius (e^r). This naturally accommodates hierarchical taxonomies.
- **Mechanism:**
  1. **Backbone:** A Transformer (like BERT) generates an initial embedding.
  2. **Exponential Map:** This embedding is projected into the Poincaré ball.
  3. **Hyperbolic Distance:** Classification uses the hyperbolic distance metric rather than Euclidean cosine similarity.

$$d_{\mathbb{D}}(u,v) = \operatorname{arccosh}\left(1 + 2\frac{\|u-v\|^2}{(1-\|u\|^2)(1-\|v\|^2)}\right)$$

- **Result:** This allows the model to distinguish between closely related categories (e.g., "Coffee Shop" vs. "Restaurant") with much higher precision than standard models, especially for "long-tail" merchants that appear infrequently. It creates a robust "Moat" against competitors using standard APIs.[20]

3.3 Anomaly Detection Engine: Isolation Forests

For the "Anomaly Detection Engine," we require an unsupervised learning approach, as "anomalies" are rare and often undefined (new fraud patterns).

- **Algorithm: Isolation Forest**. Unlike distance-based methods (K-Means) which try to group normal points, Isolation Forests explicitly isolate anomalies.
- **Logic:** Anomalies are "few and different." If we randomly partition the feature space, anomalies will be isolated in fewer cuts (shorter path length in the tree) than normal points.
- **Implementation:** We deploy this as a streaming inference model. As transactions flow through the Kafka topic, the Isolation Forest scores them. If the anomaly score exceeds a threshold, a "Flag" event is generated for the AI Accountant to review with the user.[3]

4. Infrastructure and Deployment: The Ray Ecosystem

Deployment at the scale of one million users requires an infrastructure that handles high concurrency for the API and heavy computation for the AI, without one blocking the other.

4.1 The Limits of FastAPI and the Need for Ray

While FastAPI is excellent for I/O-bound tasks, Python's Global Interpreter Lock (GIL) limits its ability to handle the CPU-intensive tasks of running the TFT and HypCD models. Scaling FastAPI workers horizontally (adding more pods) is inefficient because it duplicates the memory overhead of the ML models (which can be gigabytes in size) for every worker.

4.2 Ray Serve Architecture

We introduce **Ray Serve** as the model serving layer. Ray serves acts as a unified interface for all AI models, sitting behind the FastAPI gateway.

- **Model Composition:** Ray allows us to define a "Pipeline" of models. A single request to "Analyze Transaction" can trigger a DAG (Directed Acyclic Graph) of Ray actors:
    1. **Preprocessor Actor:** Cleans the string.
    2. **Embedding Actor (HypCD):** Computes the hyperbolic embedding.
    3. **Forecasting Actor (TFT):** Updates the user's forecast.
- **Fractional GPUs:** Ray allows us to partition a physical GPU. We can assign 0.2 of an NVIDIA A10G to the Categorization model and 0.4 to the heavy Forecasting model. This dramatically reduces cloud costs compared to dedicating a full GPU to each service.
- **Autoscaling:** Ray Serve has built-in autoscaling based on queue depth. If the "Forecasting" queue backs up during a peak period, Ray spins up more replicas of just that specific actor, utilizing the cluster resources efficiently.[23]

4.3 Serverless Training with Modal

For the "retraining" loops (e.g., updating the TFT model every week with new user data), we utilize **Modal**. Modal allows us to run Python functions in the cloud without managing infrastructure. We can spin up a high-memory GPU container for the duration of the training job and shut it down immediately after, employing a "pay-for-what-you-use" model that is far cheaper than maintaining a dedicated training cluster.[26]

5. Privacy-Preserving AI: Federated Learning

To truly "replace accountants" while maintaining trust, we must address the privacy paradox: the AI needs user data to learn, but users don't want their data exposed.

5.1 Cross-Device Federated Learning Architecture

We implement a **Federated Learning (FL)** system using the **Flower (flwr)** framework.

1. **Local Training:** The SCALE mobile app (or a lightweight local web worker) downloads the global categorization model. It trains this model *locally* on the user's device using their actual transaction history. For example, if a user re-categorizes "Amazon" from

'Shopping' to 'Business', the local model learns this preference immediately.
2. **Parameter Aggregation:** The device computes the *gradients* (the mathematical updates to the model weights) but does *not* upload the raw data.
3. **Secure Aggregation:** These gradients are sent to the central server. The server uses **Secure Multi-Party Computation (SMPC)** to sum the gradients from thousands of users. This mathematical protocol ensures that the server can see the *total* update but cannot see the *individual* update of any specific user.
4. **Global Update:** The global model is updated with the aggregated knowledge and redistributed to all users.

This architecture enables the "AI Accountant" to learn global spending patterns (e.g., "Netflix price increased") without ever centralizing the private banking data of its users, adhering to the strictest interpretation of privacy laws.[27]

6. Real-Time Communication: Event-Driven UI

The interface of the SCALE app must feel alive. When the "AI Accountant" is thinking, the user should see the thought process.

6.1 Server-Sent Events (SSE) for AI Streaming

For the agentic chat interface, we utilize **Server-Sent Events (SSE)**. Unlike WebSockets, which are bidirectional and heavy on server resources (maintaining open TCP connections with heartbeats), SSE is unidirectional (Server -> Client) and operates over standard HTTP.

- **Workflow:**
  1. User sends a prompt: "Check my financial health."
  2. FastAPI accepts the request and triggers the LangGraph agent.
  3. The agent yields intermediate steps: "Analyzing income...", "Checking fixed expenses...", "Detecting anomalies...".
  4. FastAPI streams these chunks as text events to the Next.js client.
  5. The client renders these updates in real-time, creating a "Streaming UI" that mimics human typing speed and reduces perceived latency.[30]

6.2 Next.js Data Fetching Patterns

To optimize the dashboard performance for one million users, we adopt the **Next.js App Router** with **React Server Components (RSC)**.

- **Server-Side Fetching:** The initial dashboard state (Account Balances, Monthly Spend) is fetched on the server (directly from Redis/ClickHouse) and rendered into HTML. This ensures the Fastest Contentful Paint (FCP).
- **Streaming Hydration:** Heavy components (like the "12-Month Forecast Chart" powered by TFT) are wrapped in <Suspense>. The page loads instantly with a skeleton for the chart, and the heavy data streams in parallel, "hydrating" the chart when ready. This

prevents the heavy AI computations from blocking the initial page load.[32]

7. Implementation Roadmap & Repo Refactoring

Based on the analysis of the conceptual "repo" [3], specific refactoring steps are required to support this architecture.

7.1 "Repo" Restructuring

The current monorepo structure should be evolved:

- /apps/web: The Next.js frontend (remains Vercel/Edge hosted).
- /apps/api: The new FastAPI gateway (Dockerized, deployed to K8s/Cloud Run).
- /packages/ai-engine: The Ray Serve definitions.
  - /models/tft: PyTorch implementation of Temporal Fusion Transformer.
  - /models/hypcd: Hyperbolic embedding modules.
  - /agents: LangGraph agent definitions.
- /packages/database: Prisma schema (for Postgres) and migration scripts for ClickHouse.
- /infra: Terraform/Pulumi scripts for provisioning the Ray cluster, Kafka/Redpanda, and Redis.

7.2 The "Privacy.ts" Evolution

The existing privacy.ts file should be expanded from simple anonymization functions into a full **Federated Learning Client**. It should include the flwr (Flower) client-side SDK logic to handle the download of model weights, local training loops using ONNX/TensorFlow.js (for web) or CoreML (for mobile), and the secure upload of gradients.

8. Conclusion

The architecture defined herein for the SCALE application moves beyond the traditional boundaries of web development into the realm of high-performance distributed computing. By leveraging **Event-Driven Architecture** to decouple ingestion from analysis, **Polyglot Persistence** to optimize for diverse data shapes, and **Ray Serve** to orchestrate complex **Deep Learning** pipelines, the platform can scale to one million users while delivering real-time, personalized financial intelligence.

The integration of **Hyperbolic Category Discovery** provides a distinct accuracy advantage in merchant classification, while **Temporal Fusion Transformers** offer the interpretability necessary for financial trust. Finally, by adopting the **Account Aggregator** framework and **Federated Learning**, the platform ensures that this intelligence does not come at the cost of user privacy. This is a blueprint for a resilient, scalable, and genuinely intelligent "AI Accountant."

9. Appendix: Technical Reference Tables

9.1 Database Technology Selection Matrix

Feature	**PostgreSQL (Citus)**	**ClickHouse**	**Qdrant**	**Redis**
Primary Role	Transactional Ledger (OLTP)	Analytical Engine (OLAP)	AI Memory (Vector Search)	Caching & Pub/Sub
Scaling Strategy	Sharding by user_id	Columnar Compression	HNSW Indexing	Cluster Mode
Key Use Case	Account balances, Auth	"Spend last 5 years", Trends	"Find similar transactions"	Session store, Real-time features
Write Latency	Low (Single Shard)	Medium (Batch Insert)	Medium (Index Rebuild)	Ultra-Low

9.2 Architecture Component Summary

Layer	**Technology**	**Responsibility**
Presentation	Next.js / React	UI Rendering, SSE Consumption, Client-side FL training.
Gateway	FastAPI	Request Validation, Auth, Route orchestration.
Messaging	Apache Kafka / Redpanda	Decoupling services, Event buffering, Backpressure management.
Compute	Ray Serve	Model composition, Distributed Inference,

		Autoscaling.
Training	Modal / Ray Train	Batch model retraining, Hyperparameter tuning.
Orchestration	LangGraph	Stateful agent workflows, Tool execution loops.
Privacy	Flower (flwr)	Federated Learning orchestration, Secure Aggregation.
Observability	OpenTelemetry	Distributed tracing across Python and Node.js services.

Works cited

1. Building a Scalable AI-Powered Fintech Platform: Architecture with Java Spring Boot and Amazon Bedrock - DEV Community, accessed on February 12, 2026, https://dev.to/shiv-centcapital/building-a-scalable-ai-powered-fintech-platform-architecture-with-java-spring-boot-and-amazon-25kl
2. The Future of Scalable Digital Architecture in Fintech - DevOps.com, accessed on February 12, 2026, https://devops.com/the-future-of-scalable-digital-architecture-in-fintech/
3. 2026-02-11-command-messagesuperpowersbrainstormcommand-m.txt
4. Python Performance at Scale: How We Built Systems Handling 1 Million Daily Requests, accessed on February 12, 2026, https://www.addwebsolution.com/blog/python-scale-performance
5. Ray Serve with vs without FastAPI, accessed on February 12, 2026, https://discuss.ray.io/t/ray-serve-with-vs-without-fastapi/1117
6. FIPs & FIUs in the Account Aggregator Ecosystem - BOI - Bank of India, accessed on February 12, 2026, https://bankofindia.bank.in/account-aggregator
7. Regulation of information flows as Central Bank functions? - Center on Finance, Law & Policy, accessed on February 12, 2026, https://financelawpolicy.umich.edu/sites/cflp/files/2021-10/raghavan-singh-regulation-of-information-flows-as-central-bank-functions-implications-from-treatment-account-aggregators-india.pdf
8. FAQ - Sahamati, accessed on February 12, 2026, https://sahamati.org.in/faq/
9. Masterclass on Account Aggregator Technical Architecture - Sahamati, accessed on February 12, 2026, https://sahamati.org.in/aa-hackathon-masterclass-technical-masterclass-on-acc

ount-aggregator-technical-architecture/
10. NBFC - Account Aggregator (AA) API Specification - ReBIT, accessed on February 12, 2026, https://specifications.rebit.org.in/NBFC-AA%20API%20Specification_Core_Final_08Nov.pdf
11. FinTech At Scale: How PostgreSQL Citus Handles High-Frequency Transactions - ScaleGrid, accessed on February 12, 2026, https://scalegrid.io/blog/fintech-at-scale-how-postgresql-citus-handles-high-frequency-transactions/
12. Why these three fintech companies scaled with distributed SQL - CockroachDB, accessed on February 12, 2026, https://www.cockroachlabs.com/blog/fintech-companies-scaled-distributed-sql/
13. Scale an ARIMA_PLUS univariate time series model to millions of time series | BigQuery, accessed on February 12, 2026, https://docs.cloud.google.com/bigquery/docs/arima-speed-up-tutorial
14. How do vector databases differ from relational databases? - Milvus, accessed on February 12, 2026, https://milvus.io/ai-quick-reference/how-do-vector-databases-differ-from-relational-databases
15. Qdrant - Vector Database - Qdrant, accessed on February 12, 2026, https://qdrant.tech/
16. Milvus vs Qdrant | Vector Database Comparison - Zilliz, accessed on February 12, 2026, https://zilliz.com/comparison/milvus-vs-qdrant
17. Interpretable Deep Learning for Time Series Forecasting - Google Research, accessed on February 12, 2026, https://research.google/blog/interpretable-deep-learning-for-time-series-forecasting/
18. Evaluating the Effectiveness of Time Series Transformers for Demand Forecasting in Retail, accessed on February 12, 2026, https://www.mdpi.com/2227-7390/12/17/2728
19. Temporal Fusion Transformer: A Primer on Deep Forecasting in Python, accessed on February 12, 2026, https://towardsdatascience.com/temporal-fusion-transformer-a-primer-on-deep-forecasting-in-python-4eb37f3f3594/
20. [2504.06120] Hyperbolic Category Discovery - arXiv, accessed on February 12, 2026, https://arxiv.org/abs/2504.06120
21. Hyperbolic Category Discovery - Visual AI Lab, accessed on February 12, 2026, https://visual-ai.github.io/hypcd/
22. Hyperbolic Category Discovery - CVF Open Access, accessed on February 12, 2026, https://openaccess.thecvf.com/content/CVPR2025/papers/Liu_Hyperbolic_Category_Discovery_CVPR_2025_paper.pdf
23. Ray Serve: The Versatile Assistant for Model Serving - GeekCoding101 - Make your way to geek, accessed on February 12, 2026, https://geekcoding101.com/posts/ray-serve-the-versatile-assistant-for-model-se

rving
24. Ray Serve: Scalable and Programmable Serving — Ray 2.53.0 - Ray Docs, accessed on February 12, 2026, https://docs.ray.io/en/latest/serve/index.html
25. Architecture overview - What Ray Serve LLM provides, accessed on February 12, 2026, https://docs.ray.io/en/latest/serve/llm/architecture/overview.html
26. Top Anyscale alternatives for AI/ML model deployment | Blog - Northflank, accessed on February 12, 2026, https://northflank.com/blog/anyscale-alternatives-for-ai-ml-model-deployment
27. Federated learning: what it is and how it works | Google Cloud, accessed on February 12, 2026, https://cloud.google.com/discover/what-is-federated-learning
28. What Is Federated Learning? A Guide to Privacy-Preserving AI - Palo Alto Networks, accessed on February 12, 2026, https://www.paloaltonetworks.com/cyberpedia/what-is-federated-learning
29. hey guys, is flower framework better or tensor flow federated framework? - Reddit, accessed on February 12, 2026, https://www.reddit.com/r/FederatedLearning/comments/zqvqkk/hey_guys_is_flower_framework_better_or_tensor/
30. Streaming AI Responses in Real-Time with SSE in Next.js & NestJS - WeAreDevelopers, accessed on February 12, 2026, https://www.wearedevelopers.com/videos/1630/streaming-ai-responses-in-real-time-with-sse-in-next-js-nestjs
31. Building Production-Ready SSE in Next.js: A Complete Guide | by simon ouyang | Medium, accessed on February 12, 2026, https://xiouyang.medium.com/building-production-ready-sse-in-next-js-a-complete-guide-18450fb74b7a
32. 5 Design Patterns for Building Scalable Next.js Applications - DEV Community, accessed on February 12, 2026, https://dev.to/nithya_iyer/5-design-patterns-for-building-scalable-nextjs-applications-1c80
33. Next.js SaaS Dashboard Development: Scalability & Best Practices - Ksolves, accessed on February 12, 2026, https://www.ksolves.com/blog/next-js/best-practices-for-saas-dashboards