International Journal of Latest Research in Engineering and Technology (IJLRET)
ISSN: 2454-5031
www.ijlret.com || Volume 11 - Issue 05 || May 2025 || PP. 07-12

Enterprise Scaling Strategy: Building a Resilient Metabase Architecture

Chetan Urkudkar
Senior Staff Software Development Engineer, Liveramp Inc
San Ramon, California, USA

Abstract: This study describes the construction of fault-tolerant and scalable solutions based on open-source BI platforms under rapid growth in enterprise data volume and the evolution of business analytics. The relevance of the work is determined by the forecasted doubling of the global BI-platform market by 2032 and organizations' growing need for continuous, embedded analytics with guaranteed response times, making the combination of horizontal scaling, multi-tier caching, and a fault-tolerant storage layer critical. The novelty of the research lies in the systematic integration of three directions: the evolution of Metabase's built-in cache mechanisms, advanced Kubernetes autoscaling practices (configuring HPA/VPA by the active_query_count metric and applying GitOps patterns), and optimizations of storage engines (PostgreSQL 17, Snowflake dynamic tables). The author's empirical case study—covering end-to-end Metabase integration for over one hundred organizations—confirms the practical effectiveness of the proposed approaches.

The main conclusions are: first, hybrid caching with differentiated TTLs by query type and predictive invalidation significantly reduces storage load without sacrificing interactivity; second, fine-tuned Kubernetes HPA/VPA based on Usage Analytics ensures stable replicas and optimal resource utilization; third, a PostgreSQL shared-schema model combined with a Snowflake offload layer enables both tenant-count scaling and strict data isolation, while reducing infrastructure costs by up to 50%.

This article will be helpful to engineering and DevOps teams responsible for building and maintaining high-load BI solutions on open-source platforms.

Keywords: Metabase, scaling, caching, Kubernetes, multi-tenancy, performance, BI platforms, open-source.

Introduction

Modern enterprise analytics relies on BI platforms, as they transform exponentially growing operational and behavioral data into formalized insights required for strategic decision-making. Market conditions illustrate this demand: in 2024, the global BI segment was valued at USD 31.98 billion, and by 2032, it is projected to double to USD 63.20 billion, corresponding to an 8.4% CAGR [1]. Growth is fueled by organizations' shift from discrete reports to continuous, in-workflow analytics, where not only insight depth but also speed of delivery is crucial.

In this environment, open-source solutions gain prominence: they allow rapid adaptation of the analytics stack to specific requirements without long-term licensing commitments. Empirical research shows that open-source software is already present in 96% of corporate codebases and, according to [2], saves companies 3.5 times in development costs compared to a no-open-source scenario. However, democratized data access exacerbates scaling challenges: query frequency and dataset volumes increase as the user base and data-model complexity grow. Yet interactivity expectations remain stringent: user-facing analytics UIs demand response times < 100ms, since delays over one second trigger cognitive switching and decrease engagement [3]. Thus, a BI-platform architecture must meet latency and availability SLA metrics while optimizing total cost of ownership. This study draws on the author's practice—integrating Metabase end-to-end (from data source to client dashboards) to demonstrate that a judicious combination of horizontal scaling, multi-tier caching, and a fault-tolerant storage layer can satisfy these SLAs while significantly reducing operational expenses.

Materials and Methodology

To develop a substantiated Metabase scaling strategy, 14 key sources were analyzed: the Fortune Business Insights report on BI-platform market size [1]; the SSRN study on open-source software value [2]; Tiny Bird's review of UI-latency requirements [3]; Metabase release notes for versions 48, 50 and 53 detailing cache-mechanism evolution and predictive cache layers [4–6]; Kubernetes v1.33 documentation on configurable HPA tolerances [7]; Datadog's cloud autoscaling guide [8]; PostgreSQL 17 release notes on replication and indexing optimizations [9]; the 2024 CNCF survey on Helm and GitOps practices [10]; SuperTokens' multi-tenant architecture guide [11]; Pgedge's JVM-heap tuning and VPA recommendations [12]; Snowflake documentation on dynamic tables [13]; and Metabase's built-in Usage Analytics module for performance monitoring [14].

Methodologically, the study combined:

- Comparative analysis of Metabase releases and cloud practices, correlating cache-mechanism changes (TTL invalidation at question, dashboard, and source levels [5]), predictive caching, and Snowflake integration speedups [6] with Kubernetes HPA/VPA capabilities [7, 8] and storage-engine improvements (PostgreSQL 17, Snowflake 8.x) [9, 13].
- Systematic review of orchestration and infrastructure patterns, including horizontal autoscaling by the active_query_count metric, JVM-heap optimization (Xmx ≈ 70% of cgroup limit), and GitOps deployment via Helm for configuration consistency [7, 10, 12].
- Empirical case study of end-to-end Metabase integration for over 100 organizations, where horizontal scaling, multi-tier caching, and a fault-tolerant storage layer met latency and availability SLAs while substantially lowering TCO.

Results and Discussion

Studies of Metabase scalability remain fragmented: academic publications focus on generic BI buses, while the evolution of specific open-source solutions is tracked mainly through release notes and community reports. Nevertheless, these sources allow reconstruction of optimization trends, alignment with cloud orchestration developments, and the growth of analytical database engines, providing a justified profile of the platform's current state and context for further research.

A chronological analysis of releases 48, 50, 52 and 53 shows Metabase's gradual shift from UI enhancements to deep engineering of the query path. Release 48 introduced the built-in Usage Analytics collection, initiating a "closed-loop" self-monitoring approach to identify cache bottlenecks [4] empirically. Version 50 added granular cache-policy management at the question, dashboard, and source levels, enabling a hybrid TTL-invalidation strategy [5]. In version 52 the team optimized the search ranger, tripling average search result return speed compared to version 51 ("Think 3×!" in the changelog) [6]. Finally, release 53 implemented pre-emptive caching and accelerated Snowflake synchronization by sixfold, directly targeting high-frequency source-store access latency. These changes shift the performance bottleneck from application to storage infrastructure and network layers.

In parallel, the Kubernetes community enhances vertical and horizontal autoscaling flexibility. The alpha feature, Configurable Tolerance, introduced in v1.3, allows parameterization of HPA sensitivity, damping replica-count oscillations without over-scaling [7]. For example, an HPA with a 5% scale-down tolerance and zero scale-up tolerance is configured as shown in Fig. 1:

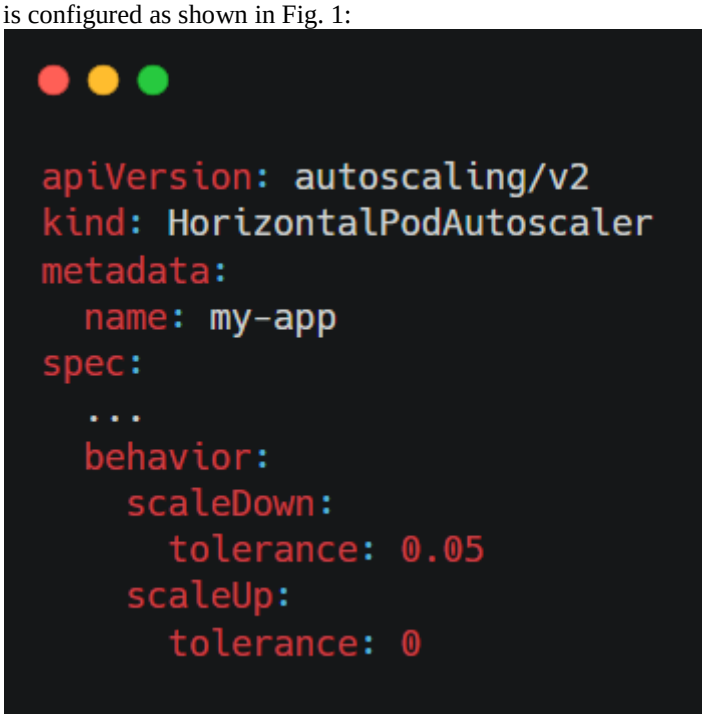

Fig. 1. HPA with a tolerance of 5% on scale-down, and no tolerance on scale-up [7]

Concurrently, Datadog reports a drop in median CPU utilization in client clusters from 16.33% to 15.9%, indicating chronic over-provisioning and underscoring VPA's role in reducing wasted resources [8]. For

Metabase—with its JVM heap spikes during syncs and complex query builds—combining HPA tuning with VPA recommendations yields a predictable platform where Usage Analytics metrics drive scaling policies.

Storage-engine progress cements these trends. PostgreSQL 17 introduced parallelized logical-replication streams and streaming I/O for sequential reads, reducing major VACUUM cycle times and boosting concurrent-write throughput [9]. In analytical scenarios, enhanced multilayer B-Tree indexing rules and ordered GROUP BY eliminate unnecessary sorts. In the cloud DWH ecosystem, Snowflake's release 8.28 adds incremental FLATTEN updates in dynamic and materialized views, cutting compute costs for aggregates that Metabase invokes tens of thousands of times daily [13]. Thus, the stack "PostgreSQL 17 / Snowflake 8.x + Metabase 53 + Kubernetes 1.33" forms a synergistic base where improved query plans, predictive caching, and adaptive autoscaling minimize overhead while preserving interactive SLAs.

Horizontal scaling of Metabase in cloud clusters begins by distributing application instances across multiple pods and enabling a horizontal autoscaler triggered by both CPU utilization and the active_query_count metric. Kubernetes v1.33's configurable tolerance for HPA lets teams set bespoke sensitivity thresholds; practical tests show that lowering the default from 10% to 5% reduces replica-count fluctuations during evening load peaks [7]. Nevertheless, Datadog continues to observe median CPU utilization of 15.9% in enterprise clusters, indicating that economic scaling gains arise not from aggressive replica proliferation but from precise threshold and limit calibration [8].

On the vertical axis, JVM tuning is critical. Further assurance comes from running the Vertical Pod Autoscaler in recommendation-only mode: VPA suggests new resource requests based on daily memory-usage dispersion. The forced application of recommendations occurs manually during scheduled releases, avoiding unexpected restarts [9].

A GitOps pattern on Helm ensures configuration-delivery reliability. According to the 2024 CNCF survey, 75% of organizations use Helm as their primary Kubernetes package manager, and 77% apply GitOps principles to some degree [10]. Helm's adoption surged from 56% in 2023 to 75% in 2024—a 33.9% annual growth rate (Fig. 2).

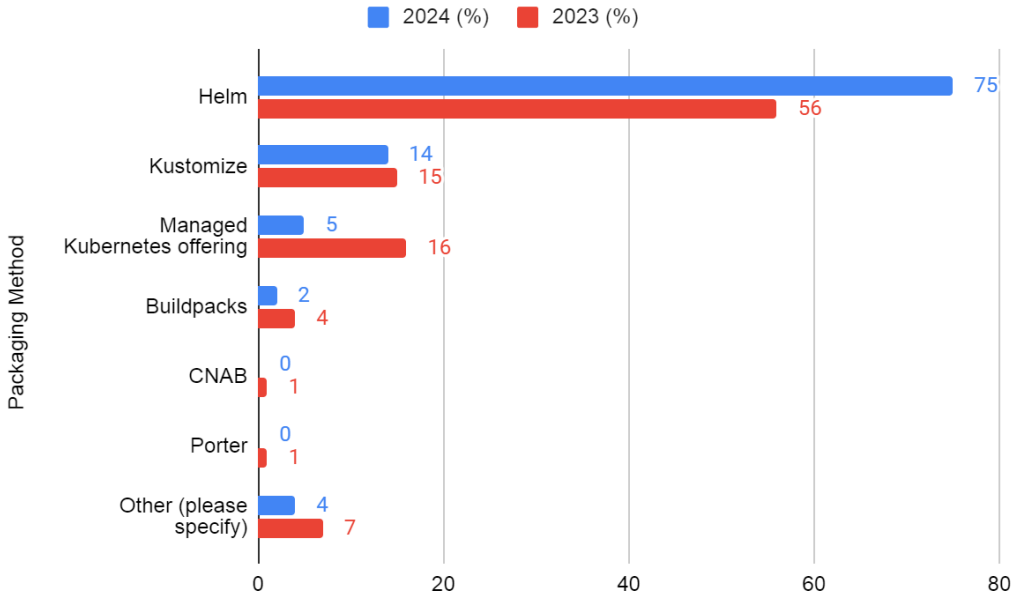

Fig. 2. The Preferred Method For Packaging Kubernetes Applications [10]

In production, all Metabase environment variables—from MB_JETTY_MAXTHREADS to connection-pool limits—are declaratively defined in values.yaml, and every change flows through pull requests, automated testing, and progressive delivery. This process minimizes configuration drift between staging and production, ensures environment reproducibility, and enables parameter-level rollbacks without downtime via rolling updates. HPA + VPA strategies, strict JVM-heap control, and right-sized HikariCP create a self-sustaining application layer capable of scaling Metabase for hundreds of tenants while maintaining latency SLAs and avoiding compute-resource waste.

Multi-tenant Metabase installations impose dual requirements on the data layer: scaling to hundreds of clients while enforcing strict data isolation. In practice, the most excellent infrastructure-cost savings come from a shared-cluster PostgreSQL model, where each tenant has its own pattern; this configuration remains

operationally manageable beyond one hundred tenants. Industry field data confirm this: companies moving from single-tenant to shared-schema multi-tenancy report up to 50% direct infrastructure-cost reduction [11]. However, dedicated databases remain essential for clients with stringent regulatory requirements (e.g., HIPAA, PCI-DSS), and a hybrid strategy—physically isolating sensitive tenants while sharing schemas for others—balances security and cost savings.

Read-load growth is handled via cascading replication: the primary node handles writes, while a reader pool distributes analytical SELECT queries. PostgreSQL 17's "failover-slot synchronization" and parallel logical-replication application eliminate slot reinitialization on leader switchovers, reducing downtime to network-retry levels [12]. Table 1 compares PostgreSQL versions.

Table 1. Comparison of different versions of PostgreSQL [12]

Feature	**PostgreSQL 15**	**PostgreSQL 16**	**PostgreSQL 17**
Logical Replication	Basic support	Improved failover recovery	Seamless failover slots
Parallel Query Support	Limited	Better parallel joins	Expanded parallel aggregates
Incremental Sort	Initial implementation	More scenarios supported	Optimized for large datasets
WAL Compression	Introduced	Improved	Faster and more efficient
Indexing	Basic deduplication	BRIN enhancements	Multi-column BRIN, better B-Tree
Autovacuum	Basic thresholds	Smarter activity-based tuning	Adaptive thresholds

Analytical workloads that scan terabyte-scale fact tables are offloaded to Snowflake. Materialized views offer a classic but costly solution; since 2024, Snowflake's dynamic tables support incremental updates: if source data remains unchanged, the service does not spin up a warehouse, resulting in effectively zero compute cost [13].

Tenant isolation is implemented at multiple levels. In PostgreSQL, row-level security is enabled for shared tables with USING policies tied to session context, but this incurs measurable overhead. Therefore, schema-per-tenant or database-per-tenant strategies remain preferable for intensive analytics with numerous joins. TLS-encrypted traffic within private subnets and Metabase-level collection-scope access attestation provides an additional security layer, minimizing the risk of cross-tenant data reads. The combination of fine-grained segregation, low-lag replication, and analytical offload creates a resilient data layer capable of serving over one hundred clients without SLA violations and with controlled cost.

Effective Metabase scaling requires quantitative control, so a baseline set of observables is first established. Exporting to Prometheus—via the built-in /metrics endpoint—captures JVM heap usage (jvm_heap_used_bytes), connection-pool activity (db_pool_active), active query count (metabase_active_query_count), and Jetty-thread parameters.

For in-system analysis, Usage Analytics auto-constructs three key dashboards. The "Performance overview" aggregates 50th and 90th percentile query times, returned row counts, and load distribution by user [14]; "Content with cobwebs" highlights questions and dashboards without views in a defined period; and a dedicated "Query log" shows running_time_seconds and cache_hit flags for each query.

Cleaning "cold" content reduces sync and cache-storage overhead. The "Clear out unused items" feature in the collections UI deletes objects not accessed, e.g., for over three months; Metabase Housekeeping recommends pre-segmentation by user groups to avoid removing niche but essential assets.

To maintain these effects, the operations team follows a three-granularity review regimen. Weekly, the Performance overview is examined, and all questions missing cache hits are tagged "optimize." Monthly, Content with cobwebs generates candidates for deletion; unused objects are moved to the trash. The Query log is aggregated quarterly by query_source to identify automated subscriptions or dashboard auto-refreshes that strain data sources, after which content owners receive recommendations to reduce refresh frequency. This cyclic procedure ties technical metrics to housekeeping actions and maintains a sustainable "signal-to-infrastructure-cost" ratio as tenant counts grow.

The author implemented end-to-end Metabase integration across the analytics stack and scaled the platform using the above mentioned strategies. The initial phase minimized processed data volumes: dashboards applied preset filters on time ranges and key dimensions, and limited the number of visualizations so that each page issued only the necessary queries. Next, a multi-tier caching mechanism was deployed in the Performance admin panel: short-TTL fixed caching for lightweight queries, scheduled invalidation for heavy queries during low-load windows, and dynamic TTL based on average execution time for intermediate queries, yielding significant latency reductions without increased RAM usage.

After stabilizing baseline performance, Usage Analytics was enabled to detail which users ran which queries, how often, the execution time, and cache efficacy. This analysis underpinned further offload: the most demanded, resource-intensive queries were moved into materialized views in the warehouse, and targeted indexes were created in PostgreSQL for hot tables and columns, reducing disk I/O and speeding aggregations. Simultaneously, content audits classified unused queries and dashboards as "cold," which were then deleted or archived to prevent excess database load and cache growth.

The combination of restrictive filters, hierarchical caching, usage analytics, and targeted storage optimization maintained interactive response times under exponential load growth typical of multi-tenant environments. Moreover, systematic removal of stale assets and focused indexation delivered measurable reductions in compute and cost overhead for Metabase, validating the practical applicability of the methods set out herein.

By employing these methods—from query-volume minimization and visualization limits to multi-tier caching, active usage monitoring, and focused index/materialized-view optimization—we establish a continuous optimization chain capable of adapting to dynamic load and operational conditions. This approach upholds interactive SLA metrics and lays the foundation for scaling Metabase in multi-client settings with minimal operational expense. Moving from ad hoc tactical tweaks to a comprehensive performance-management strategy builds a solid bridge to the generalized conclusions and recommendations presented in the Conclusion.

Conclusion

The comprehensive Metabase scaling strategy demonstrates that combining thoughtful input-query filtering, horizontal and vertical autoscaling, multi-tier caching, and targeted storage optimization yields a resilient architecture capable of withstanding exponential load growth in multi-tenant environments. Applying preliminary constraints on time ranges and key dimensions significantly reduces processed data volumes, sustaining interactive response times below 100ms and thus meeting stringent latency SLAs.

Next, multi-tier caching—split into fixed TTL for lightweight queries, scheduled invalidation for heavy queries, and dynamic TTL for intermediate cases—alleviates storage-layer load without excessive RAM consumption. Integrating Usage Analytics enables granular user-behavior insights and cache-layer efficiency assessments, redirecting the most resource-intensive and frequent queries into materialized views and indexes—thereby offloading transactional-DB load. Coupled with PostgreSQL 17 and Snowflake 8.x capabilities in parallel replication, adaptive indexing, and incremental aggregate updates, this minimizes disk operations and compute costs.

Leveraging Kubernetes v1.33 with fine-tuned HPA and VPA driven by JVM and active-query metrics creates a predictable scaling system that flexibly addresses peak loads without over-replication. For multi-tenant setups, a shared-cluster PostgreSQL model with per-tenant schemas cuts direct infrastructure costs by up to 50% while maintaining manageability; when regulatory isolation is required, sensitive tenants can be physically segregated under a hybrid strategy. Master-slave replication with parallel slot application eliminates failover downtime, and Snowflake dynamic tables prevent unnecessary compute expense when source data is unchanged. Finally, a GitOps-based Helm progressive-delivery approach ensures configuration uniformity, rapid rollback without downtime, and environment-drift control.

Thus, transitioning from disparate tactical settings to an end-to-end optimization chain—from initial data volume reduction through continuous monitoring, cold-content cleanup, and hot-query tuning—builds the foundation for a scalable, fault-tolerant, and cost-effective Metabase platform. This confirms the practical applicability of the proposed methods and establishes a basis for their extension and adaptation in future research and industrial deployments.

References

[1]. "Business Intelligence Market Leaders, Size, Share," *Fortune Business Insights*, Apr. 14, 2025. https://www.fortunebusinessinsights.com/business-intelligence-bi-market-103742 (accessed Apr. 15, 2025).
[2]. M. Hoffmann, F. Nagle, and Y. Zhou, "The Value of Open Source Software," *SSRN Electronic Journal*, 2024, doi: https://doi.org/10.2139/ssrn.4693148.
[3]. C. Archer, "User-Facing Analytics: Examples, Use Cases, and Resources," *Tiny Bird*, Apr. 23, 2024—https://www.tinybird.co/blog-posts/user-facing-analytics (accessed Apr. 06, 2025).
[4]. "Metabase 48," *Metabase*, 2024. https://www.metabase.com/releases/metabase-48 (accessed Apr. 07, 2025).
[5]. "Metabase 50," *Metabase*, 2024. https://www.metabase.com/releases/metabase-50 (accessed Apr. 07, 2025).
[6]. "Releases | What's New in Metabase," *Metabase*, 2025—https://www.metabase.com/releases (accessed Apr. 07, 2025).
[7]. "Kubernetes v1.33," *Kubernetes*, Apr. 28, 2025. https://kubernetes.io/blog/2025/04/28/kubernetes-v1-33-hpa-configurable-tolerance (accessed May 01, 2025).
[8]. N. Thomson, "Kubernetes autoscaling guide: determine which solution is right for your use case," *Datadog*, Nov. 12, 2024. https://www.datadoghq.com/blog/kubernetes-autoscaling-datadog/ (accessed Apr. 09, 2025).
[9]. "PostgreSQL: Release Notes," *PostgreSQL*, Sep. 06, 2024. https://www.postgresql.org/docs/release/17.0/ (accessed Apr. 10, 2025).
[10]. S. Hendrick, "Valerie Silverthorne, Cloud Native Computing Foundation Cloud Native 2024 Approaching a Decade of Code, Cloud, and Change," 2025. Accessed: Apr. 12, 2025. [Online]. Available: https://www.cncf.io/wp-content/uploads/2025/04/cncf_annual_survey24_031225a.pdf
[11]. "Multi-Tenant Architecture: Benefits, Practices & Implementation," *SuperTokens*, Feb. 18, 2025. https://supertokens.com/blog/multi-tenant-architecture (accessed May 05, 2025).
[12]. I. Ahmed, "PostgreSQL Performance Tuning," *Pgedge*, 2025. https://www.pgedge.com/blog/postgresql-performance-tuning (accessed Apr. 15, 2025).
[13]. "Understanding cost for dynamic tables," *Snowflake*. https://docs.snowflake.com/en/user-guide/dynamic-tables-cost (accessed Apr. 15, 2025).
[14]. "Usage analytics," *Metabase*, 2023. https://www.metabase.com/docs/latest/usage-and-performance-tools/usage-analytics (accessed Apr. 20, 2025).