Open-Source Data Management Practices and Patterns

ABHISHEK GUPTA
PRINCIPAL DEVELOPER ADVOCATE, AWS

CONTENTS

Open-source technologies form the backbone of scalable, secure, and cost-effective data solutions. The landscape offers a diverse array of options, from distributed databases to messaging systems and analytics engines. Modern data architectures often leverage multiple technologies to harness their unique strengths and features.

This Refcard explores key open-source data technologies and presents core practices for building data architectures, including infrastructure requirements, high availability strategies, optimization techniques, and more. While the core practices are broadly applicable, the examples in this Refcard focus on PostgreSQL®, Apache Cassandra®, and Apache Kafka®, providing in-depth insights into each.

OPEN-SOURCE DATA TECHNOLOGIES

Implementing open-source data architectures offers organizations significant advantages, including cost effectiveness through eliminated licensing fees, flexibility for customization, and reduced vendor lock-in. These solutions benefit from diverse community support, driving rapid innovation, frequent updates, and regular security patches.

Open-source solutions also facilitate integration with other systems, promote knowledge sharing within the developer community, and allow organizations to contribute back to the ecosystem, potentially influencing future development directions. Key categories in the open-source data ecosystem include relational and NoSQL databases, distributed messaging systems, big data processing frameworks, and data orchestration and workflow management tools.

Relational databases excel at managing structured data with predefined schemas and enforcing data integrity constraints. They're ideal for applications requiring strong consistency and sophisticated data relationships. Popular options include PostgreSQL and MySQL.

NoSQL databases are non-relational systems designed to handle specific data models, including key-value, document, column-family, and graph formats. They prioritize scalability and flexibility over strict consistency, often offering schemaless designs and distributed architectures. For example, Apache Cassandra, a column-family store, excels in write-heavy workloads and provides linear scalability across multiple nodes.

Distributed messaging systems enable real-time data streaming and event-driven architectures. They decouple data producers from consumers, providing scalable, fault-tolerant message distribution across distributed systems. Apache Kafka, for example, offers high-throughput, persistent storage of data streams, making it well suited for building large-scale data pipelines and real-time analytics applications.

instaclustr by NetApp
Learn why companies choose an open source ecosystem for data infrastructure
Learn more

instaclustr
by NetApp

Harness the power of open source

Build and scale reliable applications faster

Expert consulting and global support

- ☑ Migration to open source
- ☑ Open source strategies
- ☑ Heath checks
- ☑ Technology kickstarter package
- ☑ Solution architecture
- ☑ Operational review

Managed platform for open source technologies

Deploy, manage, and monitor all components of your data layer and related infrastructure

kafka

Cadence

cassandra

APACHE Spark

APACHE ZooKeeper

Big data processing frameworks are designed to handle and analyze massive datasets across distributed computing environments. They offer scalable, parallel processing capabilities that traditional data tools can't match. Apache Spark®, for instance, provides a unified engine for large-scale data analytics, supporting batch processing, stream processing, machine learning, and graph computation on the same distributed dataset.

Data orchestration and workflow management tools help automate and manage complex data pipelines across various systems. They coordinate tasks, handle dependencies, and ensure smooth data flow between different components of a data ecosystem. Apache Airflow®, for example, allows users to programmatically author, schedule, and monitor workflows, making it easier to manage ETL (extract-transform-load) processes and data-intensive applications.

POPULAR OPEN-SOURCE DATA TOOLS

Before exploring the core practices for building data architectures, here is a high-level overview of PostgreSQL, Apache Kafka, and Apache Cassandra.

PostgreSQL is a relational database that is known for its reliability, extensibility, and standards compliance.

- It supports a wide range of features, including complex queries, foreign keys, triggers, updatable views, and transactional integrity.
- Its architecture allows for high concurrency through the implementation of multi-version concurrency control (MVCC), which enables readers and writers to operate simultaneously without locking.

With its strong emphasis on data integrity and ability to handle large volumes of data efficiently, PostgreSQL is a popular choice for enterprise applications, geospatial systems, and complex data analytics workloads.

Apache Cassandra is a scalable, high-performance distributed NoSQL database designed to handle large amounts of data.

- It provides high availability with no single point of failure, offering support for clusters spanning multiple data centers.
- Its architecture is based on a ring design and uses consistent hashing for data distribution, allowing for linear scalability and fault tolerance.
- It offers tunable consistency levels and data replication.

Cassandra's data model is based on wide column stores, making it particularly suitable for handling time series data, sensor data, and other scenarios requiring fast writes and reads across large datasets.

Apache Kafka is a distributed event streaming platform designed for high-throughput, fault-tolerant, and scalable data pipelines.

- It utilizes a publish-subscribe model, where data is organized into topics that can be partitioned across multiple servers for parallel processing.
- Its architecture includes producers that write data to topics, consumers that read from topics, and brokers that store and serve the data.
- It maintains an ordered, immutable log of events, allowing for replay and reprocessing of data streams.
- Key features include low-latency message delivery, data persistence, stream processing, and strong guarantees for message ordering and delivery.

With its ability to handle millions of messages per second, Kafka is widely used for building real-time data streaming applications, log aggregation, and metrics collection — and is used as a backbone for microservices architectures.

OPEN-SOURCE DATA MANAGEMENT: PATTERNS AND CORE PRACTICES

Let's dive into some of the key patterns and practices for deploying and managing open-source data solutions.

SCALABILITY AND HIGH AVAILABILITY

Scalability and high availability (HA) are crucial aspects of modern data architectures. As systems grow and demand increases, the ability to scale efficiently and maintain uninterrupted service becomes crucial.

PostgreSQL offers both vertical and horizontal scaling options to enhance performance and availability. Vertical scaling involves upgrading hardware resources, while horizontal scaling can be achieved through read replicas, partitioning, and sharding. High availability is supported through built-in streaming replication, which maintains standby servers for failover and logical replication for more granular control over replication of specific tables or databases.

CORE PRACTICE
Load balancing to improve performance and availability

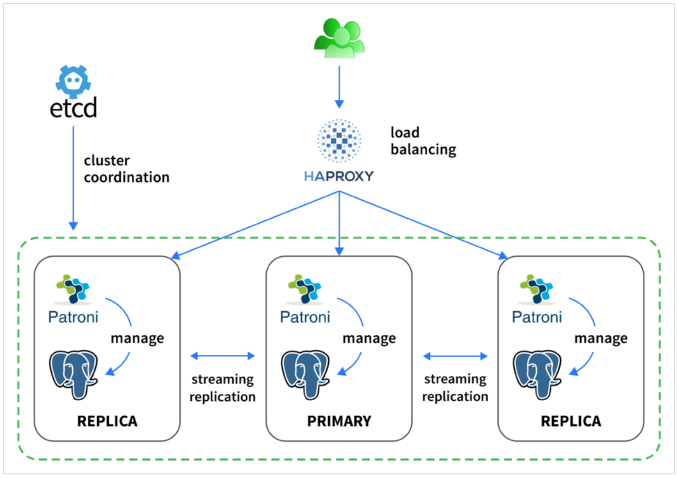
Load balancing tools like HAProxy can be used to distribute queries across multiple PostgreSQL instances.

An HA PostgreSQL setup typically involves:

- A primary server for writes
- Multiple read replicas for scaling read operations
- Streaming replication for near real-time data synchronization
- A failover mechanism (e.g., Patroni) to promote a standby to primary in case of failure
- Connection pooling and load balancing to efficiently manage client connections

This configuration provides both read scalability and HA: The primary handles writes, while read replicas serve read queries and provide failover capability.

Figure 1: PostgreSQL HA architecture

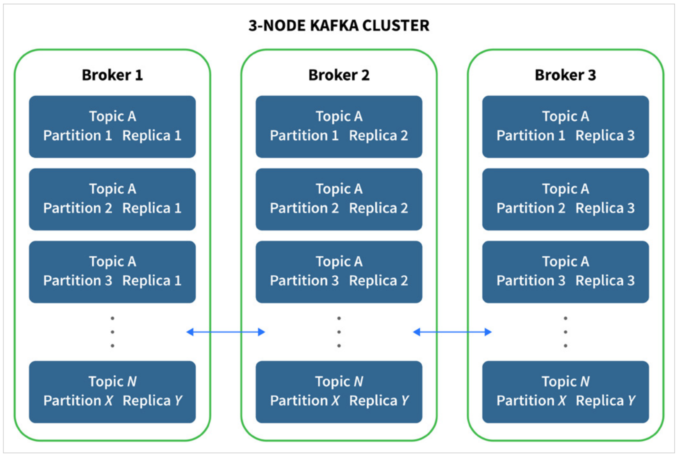


Kafka's horizontal scalability is primarily achieved through topic partitioning, allowing for distributed data processing across multiple brokers in a cluster. This architecture enables Kafka to handle high data volumes and throughput, with the ability to add brokers as needed. High availability is ensured through replication, with each partition typically having multiple replicas across different brokers.

CORE PRACTICES
• Ensure redundancy and fault tolerance by using a replication factor of three for production environments • Enhance resilience and protect against zone-level failures by deploying Kafka clusters across multiple availability zones or data centers

A typical production Kafka deployment might involve a cluster of three to five brokers spread across three availability zones, with a replication factor of three (or more) for critical topics. This setup allows for both scalability by adding more brokers or partitions and HA through replication and distribution across zones.

Figure 2: Kafka HA architecture



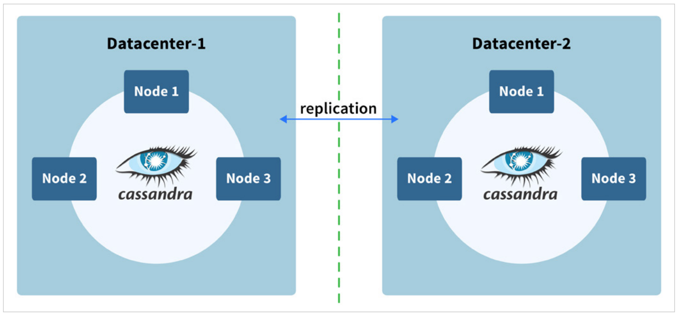
3-NODE KAFKA CLUSTER
Broker 1
Topic A Partition 1 Replica 1
Topic A Partition 2 Replica 1
Topic A Partition 3 Replica 1
Topic N Partition X Replica Y
Broker 2
Topic A Partition 1 Replica 2
Topic A Partition 2 Replica 2
Topic A Partition 3 Replica 2
Topic N Partition X Replica Y
Broker 3
Topic A Partition 1 Replica 3
Topic A Partition 2 Replica 3
Topic A Partition 3 Replica 3
Topic N Partition X Replica Y

Cassandra achieves scalability and HA through its distributed architecture. It utilizes consistent hashing to distribute data across nodes, with virtual nodes (`vnodes`) improving load balancing. Scalability is linear: Doubling node count typically doubles throughput and storage.

CORE PRACTICES
• Maintain geographic redundancy with multi-data-center replication • Allow trade-offs between read/write performance using appropriate consistency levels

A typical Cassandra deployment contains multiple data centers, each containing at least three nodes to ensure quorum-based operations. With proper configuration, this setup provides excellent scalability as nodes can be added to any data center to increase capacity. It also ensures HA, as the loss of a single node — or even an entire data center — doesn't impact overall operations.

Figure 3: Cassandra HA architecture



CAPACITY PLANNING

Effective capacity planning is crucial for ensuring optimal performance, scalability, and cost efficiency of open-source data technologies. It involves estimating hardware requirements and resource allocation to support current and future workloads.

Below are practices related to CPU, memory, network, and storage to keep in mind when working with PostgreSQL, Kafka, and Cassandra:

POSTGRESQL CORE PRACTICES	
CPU	PostgreSQL benefits from both single-thread performance and multiple cores: • For OLTP workloads, prioritize high clock speeds • For analytics or parallel query workloads, consider balancing clock speed with core count
Memory	• Allocate at least 8GB for production environments • Set `shared_buffers` to 25% of total system memory • Increase based on workload complexity, data size, and concurrent connections
Network	• Use high-bandwidth connections for write-intensive workloads or synchronous replication • In high-traffic environments, consider dedicated network interfaces for client and replication traffic
Storage	• Use SSDs for better I/O performance • Plan for 2.5-3x the raw data size to accommodate indexes, temporary files, WAL, and `vacuum` operations • Monitor usage and adjust based on workload patterns

APACHE KAFKA CORE PRACTICES	
CPU	CPU needs vary with workload: • Start with 4 cores per broker for small deployments • For large-scale production environments, consider 8-16 cores per broker • Increase for compression, encryption, or stream processing tasks • Monitor CPU utilization and scale as needed
Memory	• Start with 8GB heap space per broker • Allocate additional RAM for the page cache, typically 1-2x the heap size • For large deployments, consider up to 32GB heap • Monitor and adjust based on throughput requirements and broker performance
Network	• Use low-latency, high-bandwidth network connections • Dedicate network interfaces for inter-broker and client traffic • For multi-data-center setups, ensure robust WAN links • Monitor network throughput and latency regularly
Storage	• Use local SSDs or high-performance HDDs for the best price/performance ratio • Account for replication factor and potential traffic spikes • Monitor disk usage and adjust as needed

APACHE CASSANDRA CORE PRACTICES	
CPU	• Use multi-core processors • Start with 8 cores per node for moderate workloads, scaling to 16+ cores for write-heavy or analytics-intensive operations
Memory	• Allocate 16-32GB of RAM per node (memory is also utilized for caching, improving read performance)
Network	• Ensure high-bandwidth, low-latency connections between nodes, especially in multi-data-center deployments
Storage	• Use SSDs for optimal performance • Plan for 2-4x the raw data size to account for compaction, repair processes, and snapshots • Factor in replication strategy when calculating total cluster storage

Your requirements may vary based on workload characteristics, data models, and application needs. As your system evolves, be prepared to reassess and adjust your resource allocation. Open-source tools like Prometheus for monitoring and Grafana for visualization can provide valuable insights into resource utilization and help inform your capacity planning decisions.

DISASTER RECOVERY

While replication is crucial for HA, you should use additional **PostgreSQL** features for disaster recovery. Conduct routine tests for disaster recovery procedures, including restoration from both logical and physical backups and point-in-time recovery (PITR), in a separate environment to verify data integrity and familiarize the team with recovery processes.

CORE PRACTICES
• Enable continuous WAL archiving and use tools like `pg_basebackup`, `pg_waldump`, and `pg_receivewal` to create full backups and restore to any specific moment, protecting against logical errors and data corruption • Implement both logical (`pg_dump`) and physical (`pg_basebackup`) backups with a rotation strategy • Store backups off site or in cloud storage to safeguard against site-wide disasters

In **Kafka**-based architectures, Kafka MirrorMaker 2.0 (MM2) is used for disaster recovery solutions. It utilizes the Kafka Connect framework to perform efficient cross-cluster replication, essential for maintaining business continuity. MM2 architecture incorporates `MirrorSourceConnector` for data and metadata replication and `MirrorCheckpointConnector` for synchronizing consumer group offsets. This design enables the preservation of topic structures, partitions, and consumer offsets across clusters, facilitating rapid failover during disaster scenarios.

For effective disaster recovery, MM2 can replicate data to geographically distant locations, supporting both active-passive and active-active configurations to meet specific recovery time and point objectives. MM2 minimizes data loss and downtime in the event of cluster failures or regional outages. This ensures that in the event of a disaster, you can quickly redirect traffic to the secondary site with minimal data loss and configuration drift.

To set up MirrorMaker 2.0 for disaster recovery:

- Define the source cluster by configuring connection details for the primary Kafka cluster
- Define details for the target (backup/secondary) Kafka cluster
- Establish replication flows by specifying which topics should be replicated (use regular expressions to include or exclude specific topics)
- Configure the replication factor for mirrored topics in the target cluster
- Enable offset syncing to maintain consistency between source and target clusters

Here is an example of MM2 configuration:

```
clusters:
  primary:
    bootstrap.servers: primary-kafka:9092
 dr:
    bootstrap.servers: dr-kafka:9092

mirrors:
  primary->dr:
    source: primary
    target: dr
    topics: ".*"
    topics.exclude: "internal.*"
    replication.factor: 3
    sync.topic.acls.enabled: true
```

The configuration above specifies two clusters: `primary` and `dr` (disaster recovery), each with their respective bootstrap servers. The mirrors section establishes a replication flow from the `primary` to the `dr` cluster. It's configured to replicate all topics (`.*`) except those starting with internal (`internal.*`). The replication factor is set to `3` for mirrored topics in the target cluster, ensuring data redundancy. Additionally, the configuration enables access control list (ACL) synchronization between clusters, maintaining consistent access controls across the primary and disaster recovery environments.

Cassandra's distributed architecture provides built-in redundancy, but it's important to consider disaster recovery measures, including those below:

CORE PRACTICES	
Snapshots	Use Cassandra's snapshot feature to create point-in-time backups of your data and execute the `nodetool snapshot` command to create consistent snapshots across all nodes. These snapshots can be used for full or partial data restoration.
Incremental backups	Enable incremental backups by setting `incremental_backups: true` in `cassandra.yaml`. This creates hard links to new `SSTables`, allowing for more granular recovery options and reducing the storage overhead of full snapshots.
Commit log archiving	Configure commit log archiving by setting `commitlog_archiving_enabled: true` in `cassandra.yaml`. This allows you to capture all writes between snapshots, enabling PITR.

SECURITY

Authentication mechanisms ensure that only authorized users or applications can access your data systems, whereas perimeter security focuses on encrypting data in transit.

CORE PRACTICES

- Enforce security via common authentication measures (e.g., username/password combinations, RBAC) and consider advanced authentication methods like Kerberos
- Encrypt data in transit with SSL/TLS to prevent intrusions like eavesdropping and man-in-the-middle attacks

Authentication in **PostgreSQL** is managed through the `pg_hba.conf` file, supporting methods like password-, LDAP-, and certificate-based authentication. RBAC enables granular permission management. For encrypted connections, PostgreSQL supports SSL/TLS, which can be enabled by configuring the `ssl` parameter in `postgresql.conf`, specifying certificate and key paths, and setting `ssl=on`. Clients can then establish secure connections by using the `sslmode=require` parameter in their connection string.

For authentication, **Kafka** supports Simple Authentication and Security Layer (SASL) with mechanisms including `PLAIN`, `SCRAM`, and `Kerberos`. ACLs provide fine-grained authorization control over topics and consumer groups. To enable SSL/TLS in Kafka, you need to create a keystore with your broker's private key and certificate and a trust store with the Certificate Authority's certificate. Configure these in the `server.properties` file of each broker. Clients will need to be configured with the trust store to establish secure connections.

Example Kafka SSL configuration:

```
listeners=SSL://hostname:9093
ssl.keystore.location=/path/to/kafka.server.
keystore.jks
ssl.keystore.password=keystore-password
ssl.key.password=key-password
ssl.truststore.location=/path/to/kafka.server.
truststore.jks
ssl.truststore.password=truststore-password
ssl.client.auth=required
```

Cassandra provides several options for authentication and authorization. The default authenticator is `AllowAllAuthenticator`, which does not perform any authentication. For production environments, it's recommended to use `PasswordAuthenticator` or a custom authenticator. Cassandra's RBAC allows you to define granular permissions for users and roles. For SSL/TLS, Cassandra uses Java keystore files to store certificates. You'll need to generate a keystore for each node and a trust store for clients. Configure these in the `cassandra.yaml` file. When enabled, clients must connect using SSL to communicate with the cluster.

MONITORING

Key monitoring stats for **PostgreSQL** include query performance, connection counts, replication lag, and database size growth.

CORE PRACTICES

- Utilize `pg_stat_*` views for comprehensive database activity insights, including long-running queries (`pg_stat_activity`) and index usage (`pg_stat_user_indexes`)
- Track `vacuum` operations, especially `autovacuum` activity and table bloat, to prevent performance issues
- Monitor checkpoint frequency and duration to avoid I/O spikes
- For connection management, track active and idle connections, and if using connection pooling tools like PgBouncer, monitor their specific metrics to ensure optimal performance

Kafka's JMX metrics provide detailed insights into cluster performance, including detailed statistics on brokers, topics, and client operations. Key JMX metrics include broker-level statistics such as request rates and network processor utilization, topic-level metrics like bytes in/out per second, and consumer group metrics such as lag and offset commit rates. Tools like Prometheus with Grafana can effectively collect, visualize, and alert on these metrics, providing a holistic view of Kafka cluster health and performance.

CORE PRACTICES

- Track consumer lag to identify processing bottlenecks
- Monitor active controllers and controller changes for cluster stability
- For producers, observe batch sizes and compression ratios
- For consumers, monitor fetch request rates and sizes

Cassandra monitoring should encompass node health, cluster communication, and query performance, focusing on key metrics such as read/write latencies, compaction rates, and heap usage. Essential areas to track include gossip communication for cluster stability, pending compactions and `SSTable` counts per keyspace, read repair rates, and hinted handoffs for data consistency.

CORE PRACTICES
• Monitor tombstones and garbage collection pauses closely to prevent performance degradation and node instability • Utilize Cassandra's `nodetool` utility, particularly the `tablestats` command, for detailed insights into cluster health and table performance

TUNING AND OPTIMIZATION

PostgreSQL, Kafka, and Cassandra each offer unique configuration options for performance tuning and optimization, core practices for which are noted in the table below.

CORE PRACTICES	
PostgreSQL	• Adjust `shared_buffers` and `work_mem` for memory management • Use `EXPLAIN ANALYZE` and create appropriate indexes for query optimization • Configure `autovacuum` settings with the `vacuum` and `analyze` operations
Kafka	• Tune producer to optimize batch size and enable compression • Adjust broker configuration, including network and I/O threads and partition count
Cassandra	• Select appropriate compaction strategies based on workload characteristics • Tune memory to balance heap and off-heap memory usage • Optimize read and write paths through cache configurations and commit log settings

CONCLUSION

This Refcard explored key open-source data technologies — PostgreSQL, Apache Cassandra, and Apache Kafka — providing insights into their architectures and core management practices. We examined essential patterns for building scalable and resilient data infrastructures, including high availability strategies, capacity planning, disaster recovery, security measures, monitoring approaches, and performance optimization techniques.

By leveraging these open-source solutions and following core practices, organizations can create powerful, flexible, and cost-effective data architectures to support modern application needs.

WRITTEN BY ABHISHEK GUPTA,
PRINCIPAL DEVELOPER ADVOCATE, AWS

Over the course of his career, Abhishek has worn multiple hats including engineering, product management, and developer advocacy. Most of his work has revolved around open-source technologies, including distributed data systems and cloud-native platforms. Abhishek is also an open source contributor and avid blogger.