Migrating from Apache Solr to Amazon OpenSearch Service

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Migrating from Apache Solr to Amazon OpenSearch Service

Table of Contents

Migrating from Apache Solr to Amazon OpenSearch Service

Amazon Web Services (contributors)

January 2026 (document history)

This guide helps teams migrate from Apache Solr to OpenSearch. Solr is a search platform that's built on Apache Lucene and supports both lexical and vector search capabilities, but most teams use it primarily for lexical search. OpenSearch is a powerful search and analytics engine that's also built on Apache Lucene. Although Solr has an established community, OpenSearch has a growing developer base that frequently adds new features. This active development has led many organizations to adopt OpenSearch.

This document focuses on migrating lexical search projects from Solr to OpenSearch and covers three key areas: planning, data and metadata migration, and the proof of concept (PoC) phase.

The target audience includes search engineers, solutions architects, and migration teams that need to successfully transition their search workloads.

> **Note**
>
> OpenSearch is a fully open sourced engine. AWS offers managed services for the OpenSearch engine (Amazon OpenSearch Service and Amazon OpenSearch Serverless), as discussed in the overview. This guide uses the term *OpenSearch* interchangeably, to refer to both the search engine and and the AWS managed services that support it. In the discussion of features that are specific to the AWS services, it uses the service names.

Migrating from Solr to OpenSearch requires addressing several fundamental differences between the two products:

- **Cluster architecture**: Transitioning from the external coordination (ZooKeeper) in Solr to the self-contained cluster management in OpenSearch.
- **Data structure**: Moving from collection-based to index-centric data organization.
- **Query syntax**: Adapting queries from the parameter-based syntax in Solr to JSON query domain-specific language (DSL) in OpenSearch.

- **Configuration management**: Replacing Solr XML configurations with the REST API and YAML-based settings in OpenSearch.
- **Schema definitions**: Converting Solr `schema.xml` to mapping definitions with appropriate field type and analyzer translations in OpenSearch.
- **Scaling model**: Adopting the elastic scaling approach that OpenSearch provides instead of the shard-based scaling in Solr.
- **Security framework**: Implementing integrated security in OpenSearch instead of the plugin-based authentication in Solr.
- **Monitoring and operations**: Transitioning to Amazon CloudWatch from the Java Management Extensions (JMX) based monitoring features in Solr.

About this guide

This guide is organized into sections that address the following migration tasks.

Planning and assessment

See sections: Planning, Learning about OpenSearch

- Common migration challenges and pain points
- Cost comparison methodologies and infrastructure calculations
- Migration intake form and assessment criteria
- OpenSearch training resources and skill-building recommendations

Architectural foundation

See section: Architectural comparison

- Fundamental architectural differences between Solr and OpenSearch
- Distributed cluster management models and their implications
- Collection-based architectures compared with index-centric architectures
- Query language and operational scaling philosophies
- Comprehensive feature comparison tables

Schema and configuration migration

See sections: Migrating your schema, Migrating your configuration

- Schema migration: Detailed field, tokenizer, and filter mappings with conversion tables
- Configuration migration: Translation of Solr configuration files, including request handlers, caches, commits, and update processors

Infrastructure and security

See sections: Sizing your OpenSearch cluster, Migrating security features

- OpenSearch cluster sizing recommendations based on Solr topology
- Security considerations and implementation guidance

Data and application migration

See sections: Migrating indexing components, Migrating search queries, Migrating the Admin Console

- Data migration: Multiple approaches, including reindex strategies, connector-based tools, and custom solutions with complexity assessments
- Search migration: Query syntax conversion, SQL query support, and Learning to Rank (LTR) considerations
- Indexing component migration: Best practices for transitioning indexing workflows

Operations and monitoring

See section: Monitoring

- Administration console migration strategies
- CloudWatch-based monitoring setup and advanced monitoring options

Overview

About Solr

Apache Solr is an open source search platform that's built on Apache Lucene. It's designed to provide powerful search capabilities and enables users to search large volumes of data efficiently. Solr is highly scalable and fault-tolerant, so it's suitable for use in enterprise environments where high availability and reliability are critical.

Core features

- **Full-text search**. Solr offers robust full-text search capabilities that enable users to perform complex queries on textual data.
- **Faceted search**. Users can categorize search results into various facets, which facilitates easier navigation and filtering of results.
- **Scalability**. Solr can scale horizontally and distribute data and queries across multiple servers to handle large datasets and high query loads. Solr offers a SolrCloud deployment option that provides automated load balancing, fault tolerance, and high availability through data sharding, replication, and centralized configuration management by using ZooKeeper.
- **Real-time indexing**. Solr supports near real-time indexing, which enables new content to be searchable almost immediately after it's added.
- **Rich document handling**. Solr can index and search various document formats, including XML, JSON, and plain text.
- **Geospatial search**. Solr provides support for geospatial search, which enables users to perform location-based queries.
- **RESTful APIs**. Solr offers RESTful APIs, which make it easy to integrate Solr with other applications and services.

Use cases

- **Ecommerce search**. Solr enhances product search with features such as autocomplete, spell check, and personalized recommendations.
- **Site search**. Solr improves search functionality in websites and applications to make content more discoverable.

- **Enterprise search**. Solr provides a unified search platform across various data sources within an organization.

Deployment

You can deploy Solr in two ways: in standalone mode or in SolrCloud mode. Standalone mode is suitable for development or for small-scale applications, where a single Solr instance is sufficient. SolrCloud mode is designed for large-scale applications, where you want to distribute your workload across multiple Solr instances. SolrCloud uses ZooKeeper for cluster orchestration to ensure efficient and reliable performance. This makes SolrCloud a robust choice for handling more demanding, high-traffic environments.

Additionally, Solr supports container-based deployments, which enable it to run in Docker containers, Docker Swarm clusters, and Kubernetes environments. You can use Helm charts or operators for automated scaling and orchestration in Kubernetes.

Community and ecosystem

Apache Solr benefits from a strong community of developers and contributors who provide continuous improvements and updates. It integrates well with other big data technologies, such as Hadoop, and is often used in conjunction with data processing frameworks such as Apache Spark and Apache Kafka.

About OpenSearch

OpenSearch is fully open source search and analytics suite that's built on the Apache Lucene library. The platform includes OpenSearch (the search engine) and OpenSearch Dashboards (the visualization interface), which makes it suitable for ecommerce search, log analytics, observability, security analytics, enterprise search, and other search workloads. OpenSearch is part of the Linux Foundation. As a community-driven project, OpenSearch ensures that users maintain control over their data while they benefit from continuous improvements and security updates.

Core features

- **Full-text search**. OpenSearch provides advanced text search capabilities with support for complex queries, relevance scoring, and language analysis.
- **Distributed architecture**. OpenSearch scales horizontally across multiple nodes, which enables high availability and performance.

- **Real-time data ingestion and analytics**. OpenSearch enables rapid ingestion and analysis of large volumes of data in near real time.
- **Data visualization**. You can use OpenSearch Dashboards to create interactive visualizations and dashboards.
- **Machine learning (ML) capabilities**. OpenSearch offers anomaly detection, forecasting, and other ML-driven insights.
- **Vector database**. You can store and search billion-scale vectors that offer a variety of algorithms and engines.
- **Security features**. OpenSearch includes fine-grained access control, encryption, and audit logging.
- **Observability tools**. OpenSearch provides features for log analytics, metrics monitoring, and application performance monitoring.

Use cases

OpenSearch supports lexical search, analytics, and AI-powered semantic search. Each of these categories include multiple use cases.

Lexical search

OpenSearch excels in traditional keyword-based search scenarios. It powers critical search functionality across various industries.

- **Ecommerce search**: Enabling product discovery, faceted navigation, and personalized shopping experiences
- **Enterprise search**: Unifying search across organizational data silos and internal knowledge bases
- **Workplace search**: Facilitating employee access to documents, wikis, and corporate resources
- **Site search**: Providing fast, relevant search capabilities for websites and content platforms

Analytics

The platform's robust analytical capabilities support data-driven decision-making and operational intelligence.

- **Log analytics**: Processing and analyzing massive volumes of log data for troubleshooting and insights

- **Observability**: Monitoring application performance, infrastructure health, and system metrics
- **Anomaly detection**: Automatically identifying unusual patterns and outliers in time-series data
- **Security analytics**: Analyzing security events and threats across infrastructure for compliance and threat hunting
- **Business intelligence**: Supporting interactive dashboards and real-time analytical queries to derive actionable insights from data at scale

Generative AI and semantic search

OpenSearch supports modern AI-powered search paradigms.

- **Vector search**: Supporting dense and sparse vector embeddings for semantic understanding, with multi-modal search capabilities that combine text, images, and other data types
- **Hybrid search**: Blending traditional lexical search with vector-based semantic search for optimal relevance
- **Conversational search**: Enabling natural language interactions and query understanding
- **Retrieval Augmented Generation (RAG)**: Serving as a vector database for AI applications to retrieve contextual information
- **Model Context Protocol (MCP) integration**: Supporting MCP servers that enable AI assistants to interact with OpenSearch clusters

Deployment

OpenSearch is fully open source, so you can deploy it anywhere, by using any cloud provider. Amazon offers it as a managed service that takes care of infrastructure provisioning, installation, replication, backup, monitoring, and built-in resilience at the shard and Availability Zone level.

Managed OpenSearch is available in two deployment options:

- Amazon OpenSearch Service is a provisioned deployment option that enables you to size your cluster depending on your workload and choose your own instance types, sharding, and replication strategies. If you are looking for more control, choose Amazon OpenSearch Service.
- Amazon OpenSearch Serverless is a serverless deployment option for running OpenSearch. It automatically scales the underlying resources depending on your workload and manages sharding and data lifecycles by default.

This guide refers to OpenSearch, Amazon OpenSearch Service, and Amazon OpenSearch Serverless interchangeably.

Feature comparison

The following table compares Solr features with OpenSearch features. Some of these features are discussed in more detail later, in the Architectural comparison section.

Feature	Apache Solr	OpenSearch
Query language	Uses Apache Solr query parser and Lucene syntax; supports faceting, streaming expressions, and advanced queries.	Uses JSON-based query domain-specific language (query DSL); includes aggregations, advanced ranking, and behavior-based search. Amazon OpenSearch Service also supports SQL through the SQL plugin and the Piped Processing Language (PPL).
APIs	Provides RESTful APIs with XML and JSON support; modern JSON API is available.	Provides RESTful APIs; compatible with Elasticsearch 7.x APIs.
Deployment	Runs on Java virtual machine (JVM); supports bare metal, VMs, containers, and Kubernetes with the Solr Operator.	Provides a cloud-native focus and runs on Amazon Elastic Compute Cloud (Amazon EC2) compute or serverless resources. Provides Kubernetes support for 3.x and later releases. You can use HashiCorp Terraform or AWS CloudFormation for deployments.
Performance	Optimized for batch indexing; uses global caching per index.	Optimized for real-time indexing; caches per segment for better performance. Supports concurrent segment search, which is a performance optimization feature that parallelizes query execution at

Feature	Apache Solr	OpenSearch
		the segment level within individual shards.
Faceting and aggregations	Provides strong hierarchical faceting, which is useful for ecommerce. Supports Carrot, which is a clusterin g engine integrated into Solr that automatically organizes search results into thematic groups or categories based on content similarit y.	Includes an aggregations engine that provides control over data analysis with a high degree of freedom.
Machine learning (ML)	Provides limited ML features through streaming expressions; supports Learning to Rank (LTR).	Includes the ML Commons plugin, which provides anomaly detection, neural search, regression, RAG, and clustering.
Multi-tenancy	Uses collections to provide data isolation.	Uses indexes for multi-tenancy.
Observability	Requires external tools such as Prometheus and Grafana.	Provides built-in OpenSearch Dashboards for visualization.
Security	Supports Basic, JSON web tokens (JWTs), Kerberos authentication, audit logging, and SSL encryption.	Supports JWTs, Active Directory , LDAP, OpenID, and SAML 2.0 authentication. Can use Amazon Cognito or AWS Identity and Access Management (IAM).
Backups	Supports remote backups through Amazon Simple Storage Service (Amazon S3) and Hadoop Distribut ed File System (HDFS); provides manual snapshot management.	Provides snapshot managemen t policies for automated backups with support for various storage backends; supports cross-cluster replication. Amazon OpenSearc h Service supports automatic snapshots.

Feature	Apache Solr	OpenSearch
Replication across data centers	Uses Apache Kafka for replication across clusters.	Follows an active-passive model, where the follower index pulls data from the leader without external software, through remote reindex (`_reindex`).
Ecosystem and plugins	Supports the installation of manual extensions or using package management to include Zeppelin and Grafana.	Provides a large collection of plugins for ML, security, ingestion, and visualization.
Extract, transform, load (ETL) integration	Works with Apache NiFi and Logstash; provides ingestion pipeline support through update processors.	Integrates with Fluentd, Logstash, Vector, Data Prepper, Amazon OpenSearch Ingestion Pipeline, and other systems that can work with OpenSearch API.
Visualization	Doesn't provide built-in visualiza tion; relies on external tools such as Apache Zeppelin or Grafana.	Provides built-in OpenSearch Dashboards for real-time visualiza tion, and OpenSearch UI, which can aggregate domains under one UI.
Community and support	As a long-standing Apache Software Foundation (ASF) project, follows a slower development pace.	Is actively developed by AWS and the community with Linux Foundatio n support; provides frequent updates.
Enterprise support	Supported by third-party vendors.	Supported by AWS managed services; used in Red Hat developme nt and Pulse.
Popularity, adoption, and use cases	Popular for structured search; widely used in publishing and ecommerce.	Strong in ecommerce, log analytics , observability, vector search, and real-time applications.

Differences between Solr and OpenSearch adoption

OpenSearch and Solr are both powerful search and analytics engines that are built on Apache Lucene. Solr is an earlier search solution that has a longer history and a more mature community compared with OpenSearch. OpenSearch benefits from a vibrant and rapidly growing community that delivers cutting-edge features at a faster pace. This dynamic development environment and managed service support is driving the industry trend toward OpenSearch adoption, especially for organizations that prioritize ease of use, cloud-native advantages, and a well-rounded ecosystem. Solr remains the choice for highly customized needs and cost savings, but it requires more operational effort.

For more information, see Benefits of migrating to Amazon OpenSearch Service on the AWS Prescriptive Guidance website.

Planning your Solr migration

Migrating from Solr to OpenSearch requires careful planning because there is no direct one-to-one mapping between the two systems. Each migration requires a custom approach that's tailored to your specific use case. This guide focuses on migrating search functionality.

Although Solr and OpenSearch are both built on the Apache Lucene library, they have different architectures, deployment methods, and operational management approaches. The way applications interact with each system varies considerably. You might have built your application's search functionality by using the Solr client APIs for searching and data ingestion, but OpenSearch uses a completely different REST API structure that requires code changes.

Migrating to OpenSearch involves a steep learning curve. Without clear understanding and hands-on assistance throughout the migration process, you might face challenges because of the complexity of the process.

To determine whether OpenSearch is a viable alternative for your workload, become familiar with OpenSearch capabilities and the key challenges discussed in the next section, and consider conducting a proof of concept (PoC).

Key migration challenges

- **API and feature differences**. OpenSearch and Solr have fundamental differences with limited or no compatibility. Solr uses a file-based approach, whereas OpenSearch uses APIs to manage domains, to search, and to ingest data.
- **New features**. OpenSearch requires learning new tools, plugins, and dashboards, such as OpenSearch Dashboards and Amazon OpenSearch Service ingestion.
- **Configuration**. Setting up OpenSearch involves understanding shards, replicas, node roles, and performance tuning.
- **Data migration**. Migrating data often means reindexing, changing schemas, and checking data accuracy.
- **Security setup**. OpenSearch security requires setting up access controls, roles, and role mappings. It also involves configuring authentication backends such as basic authentication, JSON Web Tokens (JWTs), and SAML 2.0.
- **Search and indexing flow**. You will need to redesign your existing search components that are based on Solr and rewrite queries. This is because OpenSearch uses query languages such as

query domain-specific language (query DSL), Dashboards Query Language (DQL), SQL, and Piped Processing Language (PPL) instead of the query parsers that Solr uses. Additionally, you will have to redesign indexing components for compatibility with OpenSearch.

- **Change adoption**. This is the biggest challenge. Your teams might resist change because they're comfortable with existing tools, they aren't familiar with OpenSearch, or they want to avoid disruptions during the transition.
- **Custom plugins and customizations**. You will need to recreate or migrate custom Solr plugins and other customizations to work with the OpenSearch plugin architecture and APIs.
- **Feature parity gaps**. Solr features such as specialized query parsers, multi-field aliases, and the ability to update field types in place (dynamic schema capabilities) are missing in OpenSearch, so they require workarounds or alternative approaches.

For more information about these challenges, see Core architectural differences in the *Architectural comparison* section.

Start small, scale later

When you're planning to migrate to OpenSearch, you don't need a full implementation to validate functionality. A single-node cluster—even AWS Free Tier—is sufficient for your PoC. You can load a representative data sample to test your critical search patterns and use cases. This minimal setup lets you quickly determine whether OpenSearch meets your functional requirements without the complexity of multiple nodes or indexing all historical data.

After your PoC validates that all search functionality works as expected, size your workload for production: Determine the number of nodes, calculate storage for your complete dataset, and design the full cluster architecture based on performance requirements and data volumes. Testing first and scaling later saves time and money, and helps you make better infrastructure choices.

Cost comparison

When you're comparing costs, we recommend that you consider the total cost of ownership (TCO) instead of initial price differences. Although OpenSearch typically costs less than proprietary search platforms, it might appear to be more expensive than self-managed Solr. However, this initial cost comparison doesn't provide the full picture.

Amazon OpenSearch Service eliminates significant operational overhead by automatically handling hardware provisioning, software installation, patching, data backups, and infrastructure changes

through blue/green deployments. It also provides zero-downtime updates and upgrades—tasks that would otherwise require dedicated resources in a self-managed Solr environment.

These built-in capabilities substantially reduce your TCO by eliminating the time, effort, and personnel needed for routine maintenance and operations. When you factor in these operational savings, the more active OpenSearch development community, and frequent OpenSearch feature releases compared with Solr, the cost benefit becomes clear. The initial price difference is often offset by reduced operational burden and faster access to new capabilities.

Calculating infrastructure costs

Both Solr and OpenSearch use the Apache Lucene library, so their underlying data structures are similar and performance is predictable. To achieve OpenSearch performance that's similar to, or better than, the performance of your existing Solr cluster, you should size your infrastructure as follows.

1. **Document your current Solr configuration**. Capture the following key metrics from your existing Solr implementation:
   - Number of shards and replicas
   - Primary shard size
   - CPU specifications
   - Total memory and Java virtual machine (JVM) memory allocation
   - Disk capacity and type

   You can use the following Microsoft Excel spreadsheet during the initial planning phase to document your current Solr implementation.

   Download spreadsheet
2. **Select the equivalent OpenSearch infrastructure by maintaining similar resources per shard**.
   - Use the collected metrics to choose OpenSearch instances with comparable resources.
   - Reference available instance options and pricing documentation.
   - Match or exceed your current resource allocation.

   For information about available instance types and cost, see Amazon OpenSearch Service pricing.

3. **Optimize during migration.** This migration presents an opportunity to:
    - Review existing performance bottlenecks in your Solr cluster and optimize your workload.
    - Address any under-scaling issues by provisioning additional resources.
    - Enhance your infrastructure to meet or exceed desired targets.

For additional items such as buffers, indexing components, and other data movement and persistence constructs used before writing the data to Solr, see the Architectural comparison section. To calculate your actual Solr costs, you will need to factor in third-party products, managed service provider (MSP) costs, and staffing costs, which are outside the scope of this guide.

Learning about OpenSearch

Although both Solr and OpenSearch are based on Apache Lucene, they have different features and architectures. Learning the fundamentals of OpenSearch is crucial when migrating from Solr.

Both the managed Amazon OpenSearch Service and the open source version of OpenSearch have extensive documentation. AWS also offers the following educational materials to help your teams gain practical experience.

AWS Skill Builder courses

These free online courses are great for building foundational knowledge:

- Getting started with Amazon OpenSearch Service (beginner level)
- Implement lexical search using Amazon OpenSearch Service (intermediate level)

Workshops for hands-on experience

We recommend the following free workshops to help understand how OpenSearch differs from Solr, including concepts such as master nodes and operational APIs.

- Introduction to Amazon OpenSearch Service (level 100–200)
- Operational best practices for Amazon OpenSearch Service
- Dive into Amazon OpenSearch Service
- Talk to your AWS account team about attending Immersion Days

These resources are designed to guide you through the key differences between Solr and OpenSearch and provide practical experience with OpenSearch features and best practices.

Architectural comparison

This section discusses the fundamental architectural differences between Solr and OpenSearch.

Topics

Origins and governance

Apache Solr emerged in 2004 as an open source search server project under the Apache Software Foundation. After over two decades of development, Solr has evolved into a mature, community-driven platform with a stable governance model.

The Linux Foundation hosts OpenSearch under the Apache 2.0 license. Its major contributors include AWS, Red Hat, SAP, Oracle, IBM, Aiven, and other technology companies.

Core architectural differences

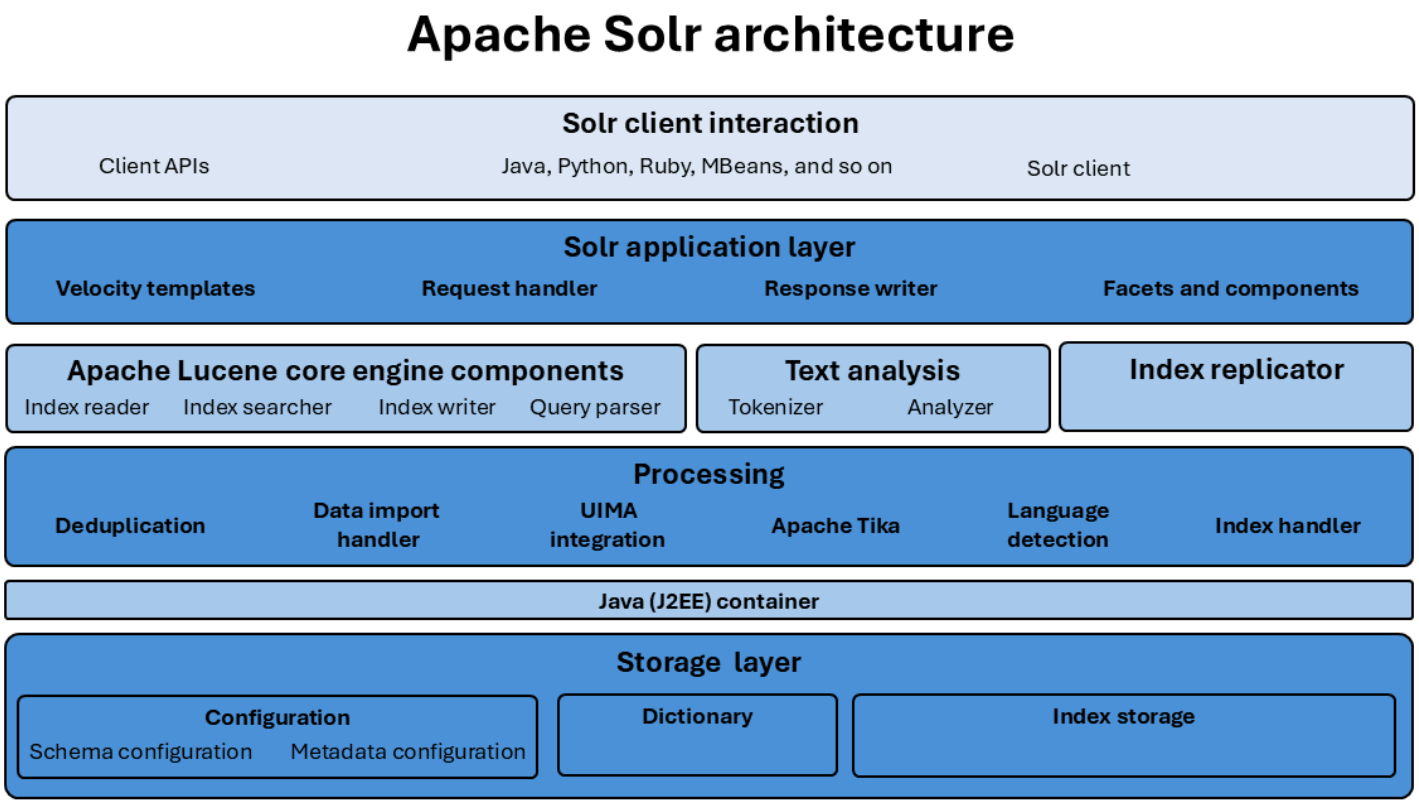
Distributed cluster management model

Solr and OpenSearch represent two distinct philosophies in managing distributed search clusters. Each provides a unique approach to distributed cluster management.

Solr takes a modular approach to distributed cluster management by using Apache ZooKeeper as an external service in SolrCloud deployments. This architecture delegates critical cluster management functions to ZooKeeper. It creates a clear separation: Solr nodes focus on search and indexing, and ZooKeeper handles the distributed orchestration.

In this model, each component has a distinct responsibility. However, the need to maintain two services introduces additional complexity and management overhead. The coordination is through a hub-and-spoke model where all Solr nodes communicate through the centralized ZooKeeper service.

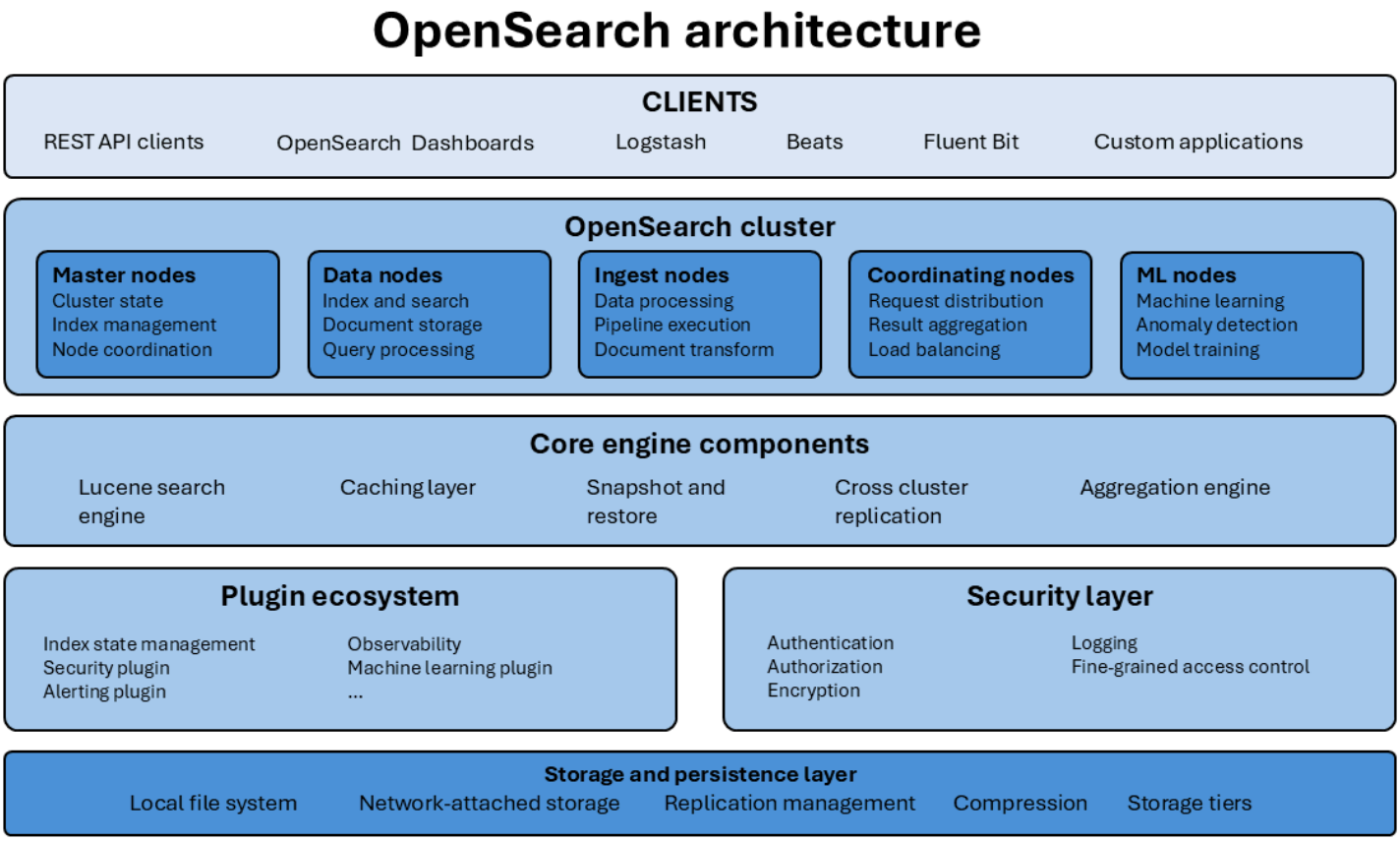
The following diagram illustrates the Solr architecture.



In comparison, OpenSearch implements an integrated distributed cluster management model. The platform uses a master-eligible node concept where designated master nodes handle cluster state management. This self-contained approach integrates cluster management with cluster state information that's published to all nodes through internal communication channels. This model simplifies deployment scenarios.

You can deploy OpenSearch clusters without additional external cluster management services such as ZooKeeper. However, this approach couples distributed cluster management with the search service, and resources are shared.

The following diagram illustrates the OpenSearch architecture.



As the architecture diagram shows, OpenSearch offers a modern, cloud-native, distributed architecture that's designed for scalability and flexibility with specialized node roles, built-in security, and extensive plugin support. Solr follows a more traditional, monolithic design that's centered around nodes. Solr architecture is simpler but less adaptable to complex, large-scale deployments. OpenSearch is architected specifically for modern, scalable cloud deployments with advanced features such as machine learning, fine-grained security, multi-tier storage, and comprehensive observability. Although Solr remains a solid choice for traditional search applications, OpenSearch is better suited for organizations that require enterprise-grade security, advanced analytics, and cloud-native scalability.

Collection and index management

Solr and OpenSearch take different approaches to organizing and managing data within their platforms, reflecting their underlying design philosophies and target use cases.

Resource abstraction

The following table compares Solr and OpenSearch components.

Component	Description	Solr	OpenSearch
Cluster	A group of nodes used for search and indexing capabilities	Uses Apache ZooKeeper for cluster orchestration.	Cluster orchestration is built in by using manager nodes (formerly known as master nodes).
Collection and index	A logical namespace that holds a complete set of searchable documents.	Typically requires a schema but also supports schemales s mode, which can automatically infer field types.	Supports dynamic mapping where fields are inferred from the data, and explicit mapping where a schema is provided at index creation.
Shard	A logical partition of an index or collection. This is a fundamental concept for achieving scalabili ty and distributed data storage.	• Consists of leader shards and replica shards. • Lacks node awareness , so leader shards and replica shards can be placed on the same node.	• Consists of primary shards and replica shards. • Enforces a strict separation of primary and replica shards on different nodes. • Amazon OpenSearch Service has Availabil ity Zone awareness that places shards in different Availability Zones when available.
Replica	A copy of the data used for redundancy.	Can be elected to a leader if the leader fails.	Can be promoted to primary if the primary fails.
Document	The fundamental unit of information	Accepts data for indexing in various formats,	Supports only JSON format for indexing.

Component	Description	Solr	OpenSearch
	that's indexed. This is equivalent to a record in a relational database.	including JSON, XML, and CSV.	
Field	Represents a specific piece of data within a document.	Fields can be configured with various parameter s within a mapping, such as index (whether the field should be indexed for searching) or store (whether the original field value should be stored).	Same as Solr.
Field type	Defines how a specific type of field data is processed.	Supported field types are covered in the Migrating your schema section.	Supported field types covered in the Migrating your schema section.
Analyzer	Transforms raw text into a structured format that the search engine can process effectively.	Defined as part of the field type.	Defined at the index level and referenced by the field. This allows multiple fields to reuse the same analyzer.

Component	Description	Solr	OpenSearch
ConfigSet	A set of configuration files that can be shared across collections.	Contains `schema.xml` , managed schema, `solrconfig.xml` , and similar files and is stored in ZooKeeper for distribution across the cluster. ConfigSets enable consistent configuration management and simplify the deployment of multiple collections that have similar requirements.	Not supported.
Collection alias	Named pointer to one or more collections.	Simplifies client access and enables collection swapping. Aliases are also useful for time-based indexes and index rotation scenarios, and provide flexibility in managing collection lifecycles.	Not supported.

Component	Description	Solr	OpenSearch
Index template	Pattern-based template for automatically configuring newly created indexes.	Not supported.	Applied when new indexes match a specified pattern. Templates enable consistent configura tion across time-series data or similar scenarios. They provide a powerful mechanism for enforcing standards and automatin g index creation.
Data stream	Time-based sequence of indexes that automatic ally creates new backing indexes.	Not supported.	Optimized for append-only time-series data such as logs or metrics. Data streams remove the complexity of managing time-based data at scale. They provide a unified interface for writing and querying while handling lifecycle management.

Component	Description	Solr	OpenSearch
Index State Management (ISM)	An automated lifecycle management system that allows you to define policies for how indexes should be managed over time based on their age, size, or document count.	Not supported.	ISM policies consist of states (such as hot, warm, cold, and delete) and transitions that define when and how indexes move between these states. This functionality enables automatic actions such as changing replica counts, moving indexes to different storage tiers, performing rollover operations, or deleting old data. It helps optimize storage costs and performance by automatically transitioning indexes from high-performance storage when they're actively written to, down to cheaper storage as they age and become read-only, and eventually deleting them when they're no longer needed.

Configuration approaches

Solr employs a file-based configuration model where core settings are defined in `solrconfig.xml` and schema definitions in `schema.xml` (or `managed-schema.xml`). In SolrCloud deployments, these configurations are stored in ZooKeeper and can be updated by

using the **upconfig** command. Although many schema modifications can be made dynamically through the Solr Schema API without requiring a restart, certain configuration changes in `solrconfig.xml`—particularly those that affect core initialization, caching, or request handlers—require a collection reload. This reload process is generally quick and doesn't require a full Solr restart, but it does momentarily interrupt query processing for that collection.

As a fully managed service for OpenSearch, Amazon OpenSearch Service takes a fundamentally different approach. You cannot access the underlying `opensearch.yml` configuration files or the server infrastructure directly. All configuration management is performed exclusively through the OpenSearch REST APIs, the AWS Management Console, the AWS Command Line Interface (AWS CLI), or infrastructure as code (IaC) tools such as AWS CloudFormation and HashiCorp Terraform.

In Amazon OpenSearch Service, you can modify most index settings, mappings, analyzers, and cluster-level configurations dynamically through these APIs without any service interruptions. Changes to index mappings, search analyzer configurations, replica counts, and numerous cluster settings are applied immediately or with a simple index close/reopen cycle. This API-driven approach provides significant operational flexibility, so you can adjust your search configurations as your requirements evolve.

Certain cluster configurations—such as instance types, storage volumes, dedicated master node settings, Availability Zone distribution, and virtual private cloud (VPC) configurations—require a blue/green deployment. During this process, a new environment is provisioned with the desired configuration, data is migrated from the old environment, and traffic is switched to the new cluster. Although this process is automated and designed to minimize downtime, it represents a more significant change operation than simple API updates.

Additionally, you don't have access to the underlying YAML configuration files, so any settings that would typically require file-level modifications in self-managed OpenSearch deployments are either exposed through AWS APIs and console options, or are not available for user modification. AWS manages security configurations, network settings, memory allocation, and other node-level parameters as part of the managed service offering.

Query language and data access differences

Solr provides flexibility through multiple query parsers that accommodate different levels of complexity and developer preferences. The traditional Solr approach uses URL parameter-based queries where search criteria, filters, sorting, and faceting are expressed as query string parameters. This method remains widely used due to its simplicity and ease of debugging directly in a browser or API testing tool. For more complex query requirements, Solr also offers a JSON request API that

structures queries in JSON format, which provides better organization for intricate search logic while maintaining relatively concise syntax. This dual approach allows developers to choose the method that best fits their use case, from simple searches to sophisticated queries with multiple clauses.

OpenSearch relies exclusively on a structured, JSON-based query domain-specific language (query DSL). All queries, regardless of complexity, are expressed as nested JSON objects that explicitly define query clauses, filters, aggregations, and other search parameters. The query DSL uses a hierarchical structure where Boolean logic, term matching, range queries, and other operations are clearly delineated within specific JSON blocks. This approach provides comprehensive expressiveness and removes ambiguity about query intent. However, it typically results in more verbose query structures compared with the URL parameter approach in Solr, even for relatively simple searches.

For more information about key architectural differences, see:

- Apache Solr vs OpenSearch – Comparison and Key Differences (BigData Boutique blog post)
- A Comprehensive Guide to OpenSearch and Elasticsearch Architecture (NetApp Instaclustr blog post)
- Karambelkar, Hrishikesh Vijay. *Scaling Apache Solr: Community Experience Distilled*. Packt Publishing Ltd., 2014.
- Apache Solr Reference Guide (Apache documentation)
- OpenSearch documentation
- Migrate from Apache Solr to OpenSearch (AWS blog post)

Operational architecture

In their scaling strategies, Solr and OpenSearch are optimized for distinct operational patterns and deployment scenarios. These differences reflect their design philosophies and target use cases in enterprise environments.

Scaling philosophy

Solr employs a scaling model that's centered on horizontal distribution through collection sharding, where data is partitioned across multiple nodes to distribute load and storage requirements. The Solr architecture maintains separate ingestion and query paths, which provide clear separation between data processing and retrieval operations.

This approach positions Solr as a dedicated search service that's typically deployed as a specialized component within larger system architectures. The separation of concerns supports the targeted optimization of search functionality, and makes Solr particularly effective in environments where search performance is the primary concern.

OpenSearch implements a more dynamic scaling approach through specialized node roles, including data nodes, coordinator nodes, and master nodes, where each type of node is optimized for specific functions within the cluster. (OpenSearch also supports ingest nodes, which aren't yet supported in Amazon OpenSearch Service.) This node role-based architecture enables elastic scaling where different aspects of the system can be scaled independently based on workload demands. The platform is designed for horizontal scaling across these specialized nodes, allowing for granular resource allocation.

The OpenSearch scaling model integrates naturally into multi-purpose data stacks that support diverse workloads beyond traditional search operations. This flexibility makes it particularly well-suited for rapid scaling in cloud environments, where resources can be dynamically allocated and deallocated based on demand. The elastic nature of the platform supports modern DevOps practices and cloud-native deployment patterns.

Both Solr and OpenSearch deliver high-performance search use cases, but their optimization strategies reflect different design priorities and target use cases.

Vector search and LLM support

Solr supports vector search capabilities through its `DenseVectorField` type and `KnnVectorQuery` functionality, and operates primarily as a self-managed solution. The Solr vector search implementation supports approximate nearest neighbor search but requires manual integration with external ML services for embedding generation. If you're running Solr on AWS, you would have to architect your own connections to Amazon SageMaker endpoints or other ML services to generate vectors, manage model versioning, and handle the operational complexity of maintaining both the search infrastructure and ML pipeline. Unlike the OpenSearch managed service approach, Solr deployments require significant operational overhead for scaling, patching, and integrating AI/ML workflows, which makes Solr less streamlined for modern vector search use cases within AWS.

As a fully managed service for OpenSearch, Amazon OpenSearch Service provides AI/ML integration through its neural search capabilities and native vector engine. The service supports k-nearest neighbors (k-NN) search by using multiple algorithms, including Hierarchical Navigable

Small World (HNSW), Inverted File Index (IVF), and brute force methods, which enable efficient similarity search across high-dimensional vector embeddings. Amazon OpenSearch Service integrates directly with Amazon SageMaker and Amazon Bedrock, so you can generate embeddings from text, images, or other data types by using pretrained large language models (LLMs) or custom ML models. The neural search plugin simplifies the ingestion-to-search pipeline by automatically vectorizing documents during indexing and queries during search time. OpenSearch also supports hybrid search approaches that combine traditional lexical search with semantic vector search by using score normalization and combination techniques. These hybrid searches provide more relevant results than either method used alone.

Plugin support

Solr provides a plugin architecture with extensibility across all core components. It supports custom request handlers, search components, update request processors, query parsers, tokenizers, and response writers through well-defined Java APIs. Solr modules include pre-built plugins for Learning to Rank (LTR), data import handlers, language detection, and clustering. You can deploy custom plugins by packaging them as JAR files and configuring them through `solrconfig.xml` or managed schemas. The plugin system lets you modify every stage in the request/response pipeline, from document indexing to query processing and result formatting. The extensibility of Solr supports its integration with external systems, implementation of custom scoring algorithms, and specialized text analysis chains for domain-specific requirements. This flexibility requires Java development expertise and careful version compatibility management during upgrades, but provides unlimited customization potential when you have specific search requirements that standard functionality cannot address.

OpenSearch provides extensibility through APIs, ingest processors, and script processors by using the Painless scripting language. The ML Commons plugin enables model hosting and inference directly within OpenSearch clusters.

As a fully managed service for OpenSearch, Amazon OpenSearch Service supports a set of plugins that are pre-installed and managed by AWS, including plugins for alerting, anomaly detection, asynchronous search, Index State Management (ISM), SQL and PPL query languages, and Performance Analyzer. The service restricts custom plugin installation to maintain security, stability, and compliance standards across the managed infrastructure. If you need custom functionality, you can submit a RFC request to the OpenSearch project for evaluation and potential inclusion in future releases. This managed approach reduces operational burden but limits the ability to deploy proprietary or experimental plugins that organizations might develop independently.

Data ingestion

Solr provides data ingestion through the Data Import Handler (DIH) framework, update request processors, and streaming expressions for pipeline construction. DIH supports direct connections to relational databases through Java Database Connectivity (JDBC). It runs SQL queries and transforms results into Solr documents without using intermediate ETL tools. Update request processors enable field manipulation, document cloning, language detection, and script-based transformations during indexing. Solr streaming expressions create computational graphs for aggregations, joins, and transformations across distributed collections. The platform accepts data through RESTful APIs in JSON, XML, and CSV formats, and SolrJ client libraries simplify application integration. Solr integrates with Apache NiFi for visual dataflow orchestration and Apache Kafka through connectors that stream records directly into collections. The Tika integration extracts text and metadata from binary documents, including PDFs and Microsoft Office files, during ingestion.

Amazon OpenSearch Service integrates with multiple AWS services for data ingestion without using traditional ETL processes. Amazon OpenSearch Ingestion (OSI) provides serverless, managed pipelines that automatically scale to handle variable data volumes from sources such as Amazon Simple Storage Service (Amazon S3), Amazon Kinesis Data Streams, Amazon DynamoDB, and Amazon Managed Streaming for Apache Kafka (Amazon MSK). These pipelines support data transformation, enrichment, and filtering by using processors such as grok, mutate, and date parsing before indexing. The service offers zero-ETL integration with Amazon S3 through direct querying capabilities, and allows federated searches across Amazon S3 data lakes without data movement. Amazon OpenSearch Service supports direct ingestion from Amazon CloudWatch Logs, AWS IoT Core, and application logs through Fluent Bit and Logstash integrations. Change data capture (CDC) from DynamoDB streams enables near real-time synchronization of database changes into OpenSearch indexes. Built-in connectors for Amazon Bedrock facilitate automatic embedding generation within the ingestion pipeline and eliminate separate vectorization workflows for AI-powered search applications.

Migrating your schema

This section explains how you can migrate your Apache Solr schema configurations to Amazon OpenSearch Service. This migration process helps you transition your Solr schema configuration to OpenSearch Index.

Prerequisites

Before you start the schema migration, make sure that you have:

- Access to your Solr schema configuration files.
- A complete understanding of your current schema usage.
- An inventory of custom field types and analyzers.
- A test environment for validation.

Topics

Solr schema

The schema for a Solr collection is defined in `schema.xml`, which contains definitions for fields and field types. The schema consists of the following elements: `fieldType`, `field`, `copyField`, `dynamicField`, `similarity`, and `uniqueKey`. The following sections describe and provide examples of schema elements. For an example of the full schema, see the appendix. For more information, see Schema Elements in the Solr documentation.

You can manage the schema in two ways: through the Schema API or through a file-based approach. When you use the API to change the schema, it automatically saves the schema to the file named `managed-schema.xml` . Alternatively, you can define the schema in a file that you can edit directly to modify the schema.

Field types

Field types define the analysis that occurs when you index data or send queries to your index. You define field types by using the `fieldType` element, which specifies the field type name, implementing class, and mandatory properties. For text fields, you specify analyzers as child elements of `fieldType` to define the text analysis process. Text analysis can occur at index time or query time.

An analyzer consists of three components:

- **Tokenizers** break field data into lexical units or tokens.
- **Filters** examine token streams and keep, transform, or discard them based on the filter type.
- **CharFilters** add, change, or remove characters while preserving original character offsets to support features such as highlighting.

In the following example, a field type named `text_general` is defined as a text field (`TextField` class). This field type defines text analysis for both indexing and querying. It uses a standard tokenizer to split text into words, followed by filters that convert tokens to lowercase, remove common stop words, and create partial word matches through NGram generation. These operations enable case-insensitive search, autocomplete functionality, and comprehensive text processing capabilities.

```
<fieldType name="text_general" class="solr.TextField">
    <analyzer type="index">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
        <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="15"/>
    </analyzer>
    <analyzer type="query">
        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
```

If you have specialized data type needs, you can create your own custom field type or customize the text analysis by creating your own customized tokenizer, filter, and so on.

In the following example, a field type named `custom_text_general` is defined as a `TextField` class. This field type specifies a custom analysis configuration for the indexing phase by using a custom tokenizer implementation (`CustomTokenizerFactory`). The custom tokenizer, which is implemented in the `com.mycompany` package, provides customized text processing logic for breaking down text into tokens during document indexing.

```
    <fieldType name="custom_text_general" class="solr.TextField">
        <analyzer type="index">
            <tokenizer class="com.mycompany.CustomTokenizerFactory"/>
        </analyzer>
    </fieldType>
```

This configuration demonstrates how Solr can be extended with custom analysis components to meet specific text processing requirements beyond the standard tokenization capabilities.

Fields

You define fields by using the `field` element, which specifies the field name and type. Each field must have a corresponding `fieldType` defined. You can identify the field type by using the `name` attribute in the `fieldType` definition. Fields can have additional properties such as `indexed`, `stored`, and `required`.

The following example defines a field named `title` that uses the `text_general` field type. This field is configured to be both indexed (searchable) and stored (retrievable in search results).

```
<field name="title" type="text_general" indexed="true" stored="true"/>
```

Copying fields

If you want to interpret document fields in multiple ways, you can use the `copyField` directive to apply different field types to a consolidated piece of incoming information.

In the following example, `copyField` directives are defined to consolidate multiple source fields into a single unified destination field named `text`.

```
    <copyField source="title" dest="text"/>
    <copyField source="description" dest="text"/>
    <copyField source="brand" dest="text"/>
    <copyField source="category" dest="text"/>
```

This configuration enables comprehensive searching across all product information through a single field. It combines title, description, brand, and category data for simplified and more powerful search capabilities.

Dynamic fields

Dynamic fields allow Solr to index fields that you did not explicitly define in your schema.

This example defines a dynamic field pattern named `attr_*`, which uses the `text_general` field type:

```
<dynamicField name="attr_*" type="text_general" indexed="true" stored="true"/>
```

This dynamic field configuration automatically creates new fields at indexing time for any field name that starts with `attr_`. It enables flexible storage and searching of varying product attributes without requiring predefined field definitions in the schema.

Unique keys

The `uniqueKey` element specifies the field that serves as a unique identifier for documents. Although this field isn't mandatory, most applications use it to control when they need to update documents in the index.

In this example, `product_id` serves as the unique field that you can use to directly update documents:

```
<uniqueKey>product_id</uniqueKey>
```

Similarity

You can use the `similarity` element to specify the class for scoring documents. You can define this element globally or within `fieldType` definitions. For example:

```
<similarity class="org.apache.lucene.search.similarities.BM25Similarity"/>
```

OpenSearch index

An index in OpenSearch is a logical namespace that organizes and stores documents. An OpenSearch index consists of two main sections: settings and mappings.

Settings

The `settings` section defines text analysis configuration: It controls how your index analyzes and processes text through analyzers, tokenizers, and filters. This section also configures index behavior and performance settings, as discussed in the OpenSearch documentation.

Example index settings:

```
{
  "sample-index": {
    "settings": {
      "index": {
        "number_of_shards": "1",
        "number_of_replicas": "1",
        "provided_name": "sample-index1"
      }
    }
  }
}
```

Amazon OpenSearch Service performs text analysis on text fields when you index a document and when you send a search request. The index analyzer processes text fields during document indexing, whereas the search analyzer processes query text during searches.

The analysis section in the settings block defines how text is processed. It consists of four components:

- `char_filter` defines character filters that process text before tokenization to prepare it for further analysis.
- `tokenizer` defines the tokenizer that receives a stream of characters and splits text into individual tokens.
- `filter` defines filters that receive token streams from the tokenizer and that add, remove, or modify tokens.
- `analyzer` defines an abstraction that encompasses text analysis. It consists of three sequentially applied components: `char_filter`, `tokenizer`, and `filter`.

Amazon OpenSearch Service supports both built-in analyzers for common use cases and custom analyzers that you can create by combining specific tokenizers, character filters, and token filters to meet specialized needs.

Mappings

The `mappings` section defines your document structure and field types. It specifies field names, types, and field mapping parameters, which are used to configure the behavior of index fields.

Example index mappings:

```
PUT /sample-index/_mapping
{
  "properties": {
    "age": {
      "type": "integer"
    },
    "occupation": {
      "type": "text"
    }
  }
}
```

OpenSearch supports dynamic mapping, which provides flexibility for different use cases.

The `properties` section in the mappings block defines how you configure individual fields.

Index templates let you dynamically initialize new indexes with predefined mappings and settings. For example, if you continuously index log data or any time-series data, you can define an index template so that all indexes have the same number of shards and replicas.

Migration flow

This section describes how you can apply an iterative approach to migrating your Solr schema to an Amazon OpenSearch Service index.

Solr and OpenSearch organize search configurations differently, but their core concepts align closely. We recommend that you fully refactor your search solution to optimize it for OpenSearch.

The migration process starts with primitive field mappings and progressively handles more complex configurations, as follows:

1. Primitive field mappings
2. Text field mappings

   a. Custom dictionary mappings
   b. Analyzer mappings
3. Custom field type mappings
4. Copy field mappings
5. Dynamic field mappings

The mappings and configurations in the following tables compare Apache Solr 9.x with OpenSearch 2.x.

Field mappings:

Solr field type class	OpenSearch field type	Analyzer support	Use case
`solr.TextField` `solr.SortableTextField`	`text`	Yes	Full-text search. For `SortableTextField` , map a `subfield` keyword with the `ignore_above` parameter set to 1000.
`solr.StrField`	`keyword`	No	Exact matching.
`solr.IntPointField`	`integer`	No	Numeric values.
`solr.LongPointField`	`long`	No	Large numbers.
`solr.FloatPointField`	`float`	No	Decimal numbers.
`solr.DoublePointField`	`double`	No	High-precision decimals.

Solr field type class	OpenSearch field type	Analyzer support	Use case
`solr.DatePointField`	`date`	No	Date/time values.
`solr.BoolField`	`boolean`	No	True/false values.
`solr.BinaryField`	`binary`	No	Binary data.
`solr.LatLonPointSpatialField`	`geo_point`	No	Geographic coordinates.
`solr.BBoxField`	`geo_shape`	No	Storing and querying complex geographic shapes.
`solr.PointType`	`xy_point`	No	N dimensional point.
`solr.NestPathField`	`nested`	No	Complex objects.
`solr.RankField`	`rank_feature`	No	Boosting or decreasing the relevance score of documents.
`solr.CurrencyField`	No direct mapping.	N/A	N/A
`solr.EnumFieldType`	No direct mapping.	N/A	N/A

Field attribute mappings:

Solr attribute	OpenSearch mapping parameter	Description	Example
`indexed="true"`	`"index": true`	Field is searchable.	Text search, filtering.
`stored="true"`	`"store": true`	Original value is stored.	Highlighting, retrieval.
`docValues="true"`	`"doc_values": true`	Field supports sorting and aggregation.	Faceting, sorting.
`multiValued="true"`	Native array support.	Field accepts multiple values.	Tags, categories.
`required="true"`	`"required": true`	Field must have a value.	Validation.
`useDocValuesAsStored="true"`	`"doc_values": true`	Use DocValues for storage.	Memory optimization.
`omitNorms="true"`	`"norms": false`	Skip scoring normalization.	Exact match fields.
`termVectors="true"`	Not supported.	Store term vectors.	Logged as unknown.
`termPositions="true"`	Not supported.	Include position information.	Logged as unknown.
`termOffsets="true"`	Not supported.	Include offset information.	Logged as unknown.

Tokenizer mappings:

Solr class	OpenSearch type	Solr parameter	Maps to
`solr.ClassicTokenizerFactory`	`standard`	`maxTokenLength`	`max_token_length` (default: 255)
`solr.KeywordTokenizerFactory`	`keyword`	`maxTokenLen`	`buffer_size` (default: 256)
`solr.LetterTokenizerFactory`	`letter`	No parameters.	N/A
`solr.LowerCaseTokenizerFactory`	`lowercase`	No parameters.	N/A
`solr.NGramTokenizerFactory`	`ngram`	`minGramSize` `maxGramSize`	`min_gram` (default: 1) `max_gram` (default: 2)
`solr.EdgeNGramTokenizerFactory`	`edge_ngram`	`minGramSize` `maxGramSize`	`min_gram` (default: 1) `max_gram` (default: 2)

`solr.PathHierarchyTokenizerFactory`	`path_hierarchy`	`reverse`
		`skip`
		`delimiter`
		`replace`
			`reverse` (default: false)
			`skip` (default: 0)
			`delimiter` (default: "/")
			`replace` (default: "/")

Solr class	OpenSearch type	Solr parameter	Maps to
solr.PatternTokenizerFactory	pattern	pattern group	pattern (default: "") group (default: -1)
solr.SimplePatternTokenizerFactory	simple_pattern	pattern	pattern (default: "")
solr.SimplePatternSplitTokenizerFactory	simple_pattern_split	pattern	pattern (default: "")
solr.StandardTokenizerFactory	standard	maxTokenLength	max_token_length (default: 255)
solr.UAX29URLEmailTokenizerFactory	uax_url_email	maxTokenLength	max_token_length (default: 255)
solr.WhitespaceTokenizerFactory	whitespace	No parameters.	N/A

Filter mappings:

Solr factory class	OpenSearch type	Solr parameter	Maps to
`solr.ASCIIFoldingFilterFactory`	`asciifolding`	`preserveOriginal`	`preserve_original` (default: false)
`solr.ApostropheFilterFactory`	`apostrophe`	No parameters.	N/A
`solr.CommonGramsFilterFactory`	`common_grams`	`ignoreCase` `words`	If `query_mode:false` (default): • `ignore_case` (default: false) • `common_words_path` (package) If `query_mode:true`, a separate filter (`solr.CommonGramsQueryFilterFactory`) is used
`solr.CJKWidthFilterFactory`	`cjk_width`	No parameters.	N/A
`solr.ClassicFilterFactory`	`classic`	No parameters.	N/A
`solr.DecimalDigitFilterFactory`	`decimal_digit`	No parameters.	N/A
`solr.EdgeNGramFilterFactory`	`edge_ngram`	`minGramSize` `maxGramSize`	`min_gram` (default: 1) `max_gram` (default: 1)

Solr factory class	OpenSearch type	Solr parameter	Maps to
		preserveOriginal	preserve_original (default: false)
solr.FingerprintFilterFactory	fingerprint	maxOutputTokenSize separator	max_output_size (default: 255) separator (default: " ")
solr.FlattenGraphFilterFactory	flatten_graph	No parameters.	N/A
solr.KeepWordFilterFactory	keep	words ignoreCase	keep_words (package) keep_words_case (default: false)
solr.KeywordMarkerFilterFactory	keyword_marker	protected	keywords_path (package)
solr.KStemFilterFactory	kstem	No parameters.	N/A
solr.LengthFilterFactory	length	min max	min (default: 0) max (default: 2147483647)
solr.LimitTokenCountFilterFactory	limit	maxTokenCount consumeAllTokens	max_token_count (default: 1) consume_all_tokens (default: false)

Solr factory class	OpenSearch type	Solr parameter	Maps to
`solr.LowerCaseFilterFactory`	`lowercase`	No parameters.	N/A

`solr.NGramFilterFactory`	`ngram`	`minGramSize`
		`maxGramSize`
		`preserveOriginal`
			`min_gram` (default: 1)
			`max_gram` (default: 2)
			`preserve_original` (default: false)

`solr.PatternReplaceFilterFactory`	`pattern_replace`	`pattern` `replacement`	`pattern` (default: "") `replacement` (default: "")
`solr.PhoneticFilterFactory`	`phonetic`	`encoder`	`encoder` (default: "metaphone")
`solr.PorterStemFilterFactory`	`porter_stem`	No parameters.	N/A
`solr.RemoveDuplicatesTokenFilterFactory`	`remove_duplicates`	No parameters.	N/A
`solr.ReverseStringFilterFactory`	`reverse`	No parameters.	N/A

Solr factory class	OpenSearch type	Solr parameter	Maps to
solr.ShingleFilterFactory	shingle	mingShingleSize maxShingleSize outputUnigrams outputUnigramsIfNoShingles tokenSeparator fillerToken	min_shingle_size (default: 2) max_shingle_size (default: 2) output_unigrams (default: true) output_unigrams_if_no_shingles (default: false) token_separator (default: " ") filler_token (default: "_")
solr.SnowballPorterFilterFactory	snowball	language	language (default: "English")
solr.StopFilterFactory	stop	ignoreCase words stopwords	ignore_case (default: false) stopwords_path (package) stopwords (default: "none")
solr.SynonymGraphFilterFactory	synonym	expand synonyms	expand (default: true) synonyms_path (package)

Solr factory class	OpenSearch type	Solr parameter	Maps to
`solr.TrimFilterFactory`	`trim`	No parameters.	N/A
`solr.UpperCaseFilterFactory`	`uppercase`	No parameters.	N/A
`solr.WordDelimiterGraphFilterFactory`	`word_delimiter_graph`	No parameters.	N/A
`solr.StemmerOverrideFilterFactory`	`stemmer_override`	dictionary	`rules_path` (package)

CharFilter mappings:

Solr factory class	OpenSearch type	Solr parameter	Maps to
`solr.HTMLStripCharFilterFactory`	`html_strip`	No parameters.	N/A
`solr.MappingCharFilterFactory`	`mapping`	`mapping`	`mappings_path` (package) `mappings` (array)

Solr factory class	OpenSearch type	Solr parameter	Maps to
`solr.PatternReplaceCharFilterFactory`	`pattern_replace`	`pattern`	`pattern` (required)
		`replacement`	`replacement` (default: "")

The following sections describe these mappings in detail and also explain how OpenSearch automatically handles unique keys and similarity search configurations.

Step 1. Map primitive fields

Start by analyzing your Solr schema, and focus first on straightforward field mappings. This creates a foundation for fields that have more complex transformations. OpenSearch has a simpler field configuration than Solr and handles many Solr field attributes automatically without explicit configuration.

For each field, identify the field name (such as `product_id`, `price`), which serves as the field identifier, the field type reference (such as `string`, `float`), and any field attributes, such as `indexed` or `stored` properties. After you identify these field components, identify the referenced field type, and map the Solr field type to its OpenSearch type equivalent.

Map Solr field attributes to OpenSearch field mapping parameters only when necessary. For example, OpenSearch fields are indexed by default, so you don't have to map the Solr attribute `indexed="true"` explicitly.

The following example demonstrates the migration of the Solr fields named `product_id`, `price`, `category`, and `brand` to Amazon OpenSearch Service. It shows how `solr.StrField` maps to `keyword` and `solr.FloatPointField` maps to `float`.

Solr basic fields:

```
<!--field types -->
<fieldType name="string" class="solr.StrField"/>
<fieldType name="float" class="solr.FloatPointField"/>

<!--fields -->
<field name="product_id" type="string" indexed="true" stored="true"/>
```

```
<field name="price" type="float" indexed="true" stored="true"/>
<field name="category" type="string" indexed="true" stored="true"/>
<field name="brand" type="string" indexed="true" stored="true"/>
```

After mapping to Amazon OpenSearch Service:

```
{
  "mappings": {
    "properties": {
      "product_id": {"type": "keyword"},
      "price": {"type": "float"},
      "category": {"type": "keyword"},
      "brand": {"type": "keyword"}
    }
  }
}
```

Step 2. Map text fields

In this step, you identify text fields that require text analysis and field types that have the `analyzer` element defined.

In the following example, you'll migrate the field named `title`. This field uses the `text_general` type, which has two analyzers defined.

```
<!-- Solr Text Field with Analysis -->
<fieldType name="text_general" class="solr.TextField">
    <analyzer type="index">
       <tokenizer class="solr.StandardTokenizerFactory"/>
       <filter class="solr.LowerCaseFilterFactory"/>
       <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
       <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="3"/>
     </analyzer>
     <analyzer type="query">
       <tokenizer class="solr.StandardTokenizerFactory"/>
       <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
       <filter class="solr.LowerCaseFilterFactory"/>
     </analyzer>
</fieldType>

<field name="title" type="text_general" indexed="true" stored="true"/>
<field name="description" type="text_general" indexed="true" stored="true"/>
```

Review the built-in analyzers in the OpenSearch documentation to find the best match.

Use built-in OpenSearch analyzers if they provide similar functionality. However, direct one-to-one mappings might not exist between Solr and OpenSearch components. Create custom analyzers if the built-in options don't meet your requirements.

Mapping custom dictionaries

Your Solr analyzers might depend on external files (such as `stopwords.txt` or `synonyms.txt`). You'll need to handle these dependencies when migrating to Amazon OpenSearch Service.

You have two options for handling custom dictionaries in Amazon OpenSearch Service: inline and by uploading files.

Inline configuration: Include word lists directly in your index settings.

```
"filter": {
        "custom_stop": {
          "type": "stop",
          "stopwords": ["the", "is", "at", "which", "on"]
        },
 }
```

Uploading files: This is the option we recommend. You can upload custom dictionary files, such as `stopwords.txt` and `synonyms.txt`, and associate them with your domain. Create custom packages by copying your Solr dictionary files to an S3 bucket, and then create an OpenSearch package and associate it with your domain. After you associate a file with a domain, you can use it in parameters such as `synonyms_path` and `stopwords_path`:

```
"filter": {
        "custom_stop": {
          "type": "stop",
          "stopwords_path": "analyzers/Fxxxxxxx"
        },
}
```

Mapping analyzers

To create a custom analyzer, identify the `tokenizer`, `filter`, and `charFilter` sections under the `analyzer` element in your Solr `fieldType` element.

To migrate your text field analyzers, identify the analyzer configuration for your field type and determine whether your field type has distinct analyzers for indexing and querying. Separate the index and query analyzer configurations and document all components within each analyzer. For each analyzer, carefully examine the configuration to identify the tokenizer, filters, and character filters.

Map each tokenizer, filter, and character filter to its OpenSearch equivalent. For example, `StandardTokenizerFactory` maps to the OpenSearch standard tokenizer, and `LowerCaseFilterFactory` maps to the OpenSearch `lowercase` filter. For detailed component mapping information, see the tables earlier in this section.

Establish a predictable naming strategy that combines the field type name with the analyzer type by using the format `{field_type_name}_{analyzer_type}`. For example, your index analyzer becomes `text_general_index` and your query analyzer becomes `text_general_search`. You can then refer to the analyzer in OpenSearch fields by using the `analyzer` or `search_analyzer` field mapping parameters.

The following example demonstrates an Amazon OpenSearch Service index with custom analysis settings for text fields. The configuration includes two analyzers: `text_general_index` for indexing and `text_general_search` for searching. Both analyzers use a standard tokenizer with custom filters. The index analyzer includes lowercase conversion, custom stop words referenced from OpenSearch packages, and N-gram filtering with token sizes ranging from 2 to 3 characters. The search analyzer uses only lowercase and stop word filtering to process queries more efficiently.

In the `mappings` section, both `title` and `description` fields are configured as text types with distinct analyzer settings. The `analyzer` parameter specifies `text_general_index` for processing text during document indexing, and the `search_analyzer` parameter specifies `text_general_search` for processing search queries:

```
{
  "settings": {
    "analysis": {
      "analyzer": {
        "text_general_index": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "custom_stop",
            "ngram_filter"
```

```
          ]
        },
        "text_general_search": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "custom_stop",
            "lowercase"
          ]
        }
      },
      "filter": {
        "custom_stop": {
          "type": "stop",
          "stopwords_path": "analyzers/FXXXXXXX"
        },
        "ngram_filter": {
          "type": "ngram",
          "min_gram": 2,
          "max_gram": 3
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "text_general_index",
        "search_analyzer": "text_general_search"
      },
      "description": {
        "type": "text",
        "analyzer": "text_general_index",
        "search_analyzer": "text_general_search"
      }
    }
  }
}
```

Validating text analysis

After you create your analyzer configuration in OpenSearch, validate that it works as expected before you index large amounts of data. Amazon OpenSearch Service provides the `_analyze` API to test your analyzers:

```
POST /your-index/_analyze{
  "analyzer": "text_general_index",
  "text": "The Quick Brown Fox Jumps"
}
```

Step 3. Map custom field types

To convert Solr custom fields to OpenSearch, evaluate whether OpenSearch native features can achieve the desired functionality before you consider custom development.

In the following example, you'll migrate the field named `custom_title`, which uses the `custom_text_general` type. This `fieldType` uses a custom implementation of the tokenizer `com.mycompany.CustomTokenizerFactory`.

```
<!-- Custom Field Types -->
<fieldType name="custom_text_general" class="solr.TextField">
    <analyzer type="index">
        <tokenizer class="com.mycompany.CustomTokenizerFactory"/>
    </analyzer>
</fieldType>
<field name="custom_title" type="custom_text_general" indexed="true" stored="true"/>
```

To migrate custom field types from Solr to OpenSearch, you can choose from two approaches: using OpenSearch built-in tokenizers and analyzers, or developing custom plugins.

The preferred option is to use OpenSearch built-in tokenizers and analyzers, which you can configure through JSON settings. This involves creating a custom analyzer definition that combines existing components such as tokenizers, token filters, and character filters to achieve the desired text analysis behavior. For example, you might use the pattern tokenizer with specific patterns, combine it with lowercase filters, or use other built-in components to replicate your custom Solr custom tokenizer's functionality.

We recommend that you consider the second option only if OpenSearch built-in components don't meet your requirements. This involves creating a custom plugin that implements your

custom tokenizer text analysis logic, and installing the plugin in OpenSearch. The plugin approach requires more development effort and ongoing maintenance but provides maximum flexibility for implementing complex text analysis logic.

To choose between these options, consider factors such as maintenance overhead, performance requirements, and the complexity of your text analysis. We recommend that you thoroughly evaluate whether the rich set of built-in analysis components in OpenSearch can meet your requirements before you develop a custom plugin.

The following example demonstrates an Amazon OpenSearch Service index with a custom text analyzer configuration. The configuration includes a single custom analyzer named `custom_text_analyzer` that uses a specialized tokenizer defined as `custom_tokenizer`. In the `mapping` section, a field named `custom_title` is configured as a text type with the custom analyzer setting. The `analyzer` parameter specifies `custom_text_analyzer` for processing text during both document indexing and search operations.

```
// Index settings
PUT /my_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_text_analyzer": {
          "type": "custom",
          "tokenizer": "custom_tokenizer"
        }
      }
    }
  }
}

// Field mapping
PUT /my_index/_mapping
{
  "properties": {
    "custom_title": {
      "type": "text",
      "analyzer": "custom_text_analyzer"
    }
  }
}
```

Step 4. Map copy fields

When you convert your Solr schema to Amazon OpenSearch Service, you can implement the `copyField` directive by using the OpenSearch copy_to parameter.

For example, the following Solr elements:

```
<!-- Unified search field - copy multiple fields to one destination -->
<copyField source="title" dest="text"/>
<copyField source="description" dest="text"/>
<copyField source="brand" dest="text"/>
<copyField source="category" dest="text"/>
```

are converted to:

```
"title": {
  "type": "text",
  "copy_to": "title_sort"
}
```

Step 5. Map dynamic fields

Amazon OpenSearch Service implements dynamic fields by using dynamic templates, which match field patterns that are similar to `dynamicField` in Solr.

For example, the following Solr element:

```
<dynamicField name="attr_*" type="text_general"/>
```

transforms into a dynamic template in OpenSearch:

```
"dynamic_templates": [{
  "attributes": {
    "match": "attr_*",
    "mapping": {
      "type": "text",
      "analyzer": "text_general"
    }
  }
}]
```

This pattern-based mapping automatically applies specified settings to any new field that matches the pattern, so it maintains the same flexible schema behavior as Solr dynamic fields.

Handling unique keys

Amazon OpenSearch Service and Solr handle unique identifiers differently. In Solr, `<uniqueKey>product_id</uniqueKey>` requires explicit configuration, whereas OpenSearch automatically provides a unique identifier through its `_id` field for each document. You can still use the `product_id` field value as the document's `_id` when you index documents.

Handling similarity configurations

In Solr, the similarity configuration controls scoring algorithms for search relevance. This feature maps to the similarity settings in OpenSearch. Amazon OpenSearch Service uses BM25 as the default ranking framework, but it supports other similarities such as Boolean as well.

Best practices

Migrating from Solr to Amazon OpenSearch Service offers a straightforward path through one-to-one mapping of fields, analyzers, and configurations. It also presents a valuable opportunity to reassess and optimize your search infrastructure.

Instead of lifting and shifting your existing Solr configurations, we recommend that you take the time to evaluate each field's necessity, validate data types for optimal performance, and simplify complex configurations where possible.

Consider whether custom Solr field types could be replaced with OpenSearch native functionality. This strategic approach not only ensures a successful migration but also takes advantage of the strengths in Amazon OpenSearch Service to help you build a more efficient, maintainable search solution. The goal isn't only to replicate Solr's functionality, but to enhance your search capabilities while reducing unnecessary complexity.

Migrating your configuration

This section describes how to migrate your Apache Solr configuration (`solrconfig.xml`) to index settings and equivalent features in Amazon OpenSearch Service.

Prerequisites

Before starting the configuration migration, make sure that you have:

- Access to your Solr configuration file (`solrconfig.xml`).
- A complete understanding of your current configuration settings.
- An inventory of your Solr request handlers and search components.
- Knowledge of your commit and performance settings.
- A test environment with Amazon OpenSearch Service deployed for validation.

Topics

Solr configuration

The configuration for a Solr collection is defined in `solrconfig.xml`, which contains configuration settings that determine index location and formatting, caching, codec factory, circuit breaks, commits, transaction logs (tlogs), query performance, request handlers, update processing chains, and other settings.

You can manage the configuration in two ways: through the Config API or through a file-based approach. When you use the API to change the configuration, it automatically creates *configuration overlays* (`configoverlay.json`) to override the values in `solrconfig.xml`. Alternatively, you can define the configuration in a file that you can edit directly to modify settings.

Key configuration elements

The following table describes Solr configuration elements and provides examples of each.

Element	Purpose	Example

`lib`	Loads custom plugins (collection-level or cluster-level).	`<!-- Lib directive Config -->`
		`<lib dir="./lib" />`
		`<lib dir="${solr.install.dir:../../../..}/modules/extraction/lib" regex=".*\.jar" />`
		`<lib dir="${solr.install.dir:../../../..}/modules/clustering/lib/" regex=".*\.jar" />`
		`<lib dir="${solr.install.dir:../../../..}/modules/langid/lib/" regex=".*\.jar" />`
		`<lib dir="${solr.install.dir:../../../..}/modules/ltr/lib/" regex=".*\.jar" />`
		`<lib path="${solr.install.dir:../../../../..}/modules/custom/lib/CustomPriceTaxProcessorFactory.jar"/>`

`dataDir` and `directoryFactory`	Configures index storage location and I/O implementation.	`<!-- Data Directory Config -->` `<dataDir>${solr.data.dir:}</dataDir>`

`codecFactory`	Controls index compression for storage efficiency.	`<!-- Codec Factory -->`
		`<codecFactory class="solr.SchemaCodecFactory">`
		`<str name="compressionMode"BEST_COMPRESSION</str>`
		`</codecFactory>`

`schemaFactory`	Specifies schema management mode: managed (API-editable) or classic (file-based).	`<!-- Schema Factory -->`
		`<schemaFactory class="ManagedIndexSchemaFactory">`
		`<bool name="mutable">true</bool>`
		`<str name="managedSchemaResourceName">managed-schema</str>`
		`</schemaFactory>`

Element	Purpose	Example

`indexConfig`	Configures segment buffer sizes, merge policies, and merge scheduler threads.	`<!-- Index Config -->`
		`  <indexConfig>`
		`  <ramBufferSizeMB>100</ramBufferSizeMB>`
		`  <maxBufferedDocs>1000</maxBufferedDocs>`
		`  <useCompoundFile>false</useCompoundFile>`
		
		`  <mergePolicyFactory class="org.apache.solr.index.TieredMergePolicyFactory">`
		`  <int name="maxMergeAtOnce">10</int>`
		`  <int name="segmentsPerTier">10</int>`
		`  </mergePolicyFactory>`
		
		`  <mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler">`
		`  <int name="maxMergeCount">7</int>`
		`  <int name="maxThreadCount">3</int>`
		`  </mergeScheduler>`
		`  </indexConfig>`

`updateHandler`	Configures hard commits (durability) and soft commits (near real-time visibility).	`<!-- updateRequestProcessorChain -->`
		`  <updateRequestProcessorChain name="standard">`
		`  <processor class="solr.TrimFieldUpdateProcessorFactory"/>`
		`  <processor class="com.mycompany.CustomPriceTaxProcessorFactory"/>`
		`  </updateRequestProcessorChain>`

Element	Purpose	Example
`query`	Configures cache settings (filter, query result, document) and query limits.	`<!-- Query Settings --> <slowQueryThresholdMillis>-1</slowQueryThresholdMillis> <maxBooleanClauses>1024</maxBooleanClauses> <enableLazyFieldLoading>true</enableLazyFieldLoading> <useColdSearcher>false</useColdSearcher> <maxWarmingSearchers>2</maxWarmingSearchers> </query>`
`requestHandler`	Configures endpoint definitions with defaults, appends, and invariants.	`<!-- Request Handlers --> <requestHandler name="/select" class="solr.SearchHandler"> <lst name="defaults"> <str name="echoParams">explicit</str> <int name="rows">10</int> <str name="df">_text_</str> </lst> </requestHandler>`
`searchComponent`	Configures modular components for spellcheck, suggestions, highlighting, and so on.	`<!-- Search Components --> <searchComponent name="spellcheck" class="solr.SpellCheckComponent"/> <searchComponent name="suggest" class="solr.SuggestComponent"/> <searchComponent name="elevator" class="solr.QueryElevationComponent"/> <searchComponent class="solr.HighlightComponent" name="highlight"/>`

Element	Purpose	Example
`updateRequestProcessorChain`	Documents transformation pipeline before indexing.	`<!-- updateRequestProcessorChain --> <updateRequestProcessorChain name="standard"> <processor class="solr.TrimFieldUpdateProcessorFactory"/> <processor class="com.mycompany.CustomPriceTaxProcessorFactory"/> </updateRequestProcessorChain>`

For detailed configuration options, see Configuring solrconfig.xml in the Solr documentation.

Migration flow

This section describes an iterative approach to migrating your Solr configuration to Amazon OpenSearch Service. We recommend that you approach the process systematically by organizing your migration into three focus areas:

1. Settings: Translating Solr configuration to OpenSearch cluster or index settings
2. Search functionality: Translating Solr search functionality to OpenSearch search functionality
3. Document processing: Translating Solr document processing to Amazon OpenSearch Ingestion (OSI)

Step 1. Migrate settings

The components of your Solr configuration map to different Amazon OpenSearch Service features. The following sections cover the migration of Solr shards and replicas, codecs, commits, caches and queries, and index segments and merging. Amazon OpenSearch Service manages index storage locations directly, so Solr `dataDir` settings don't need to be configured.

Before you migrate every configuration setting from Solr, assess whether the setting can be adjusted based on your current search system experience and best practices. Some settings are configured at the cluster or node level, and not at the index level. These include the maximum number of clauses in a Boolean query, circuit breaker settings, and cache settings.

Instead of just lifting and shifting existing configurations, take the time to evaluate each setting's necessity and simplify complex configurations where possible. For example, the slow logs threshold of one second might be intensive for logging and can be revisited. You might also want to review and reduce the `max.booleanClauses` setting.

Shards and replica settings

Solr allows replica shards to be distributed across nodes for redundancy and fault tolerance, but it lacks node awareness during shard placement. This means that primary and replica shards might inadvertently be allocated to the same node, which compromises the intended high availability and benefits of replication.

Amazon OpenSearch Service ensures that primary and replica shards are not on the same node. When you enable zone awareness, Amazon OpenSearch Service makes a best effort to distribute primary shards and their corresponding replica shards to different Availability Zones. When you configure dedicated master nodes and zone awareness for standby replicas, Amazon OpenSearch Service ensures that primary and standby replica shards are placed in different Availability Zones, which provides stronger resilience guarantees.

The OpenSearch and Solr definitions of a replica are different. In OpenSearch, you define a primary shard count by using the `number_of_shards` setting, which determines the partitioning of your data. You then set a replica count by using the `number_of_replicas` setting. Each replica is a copy of all the primary shards. Therefore, if you set `number_of_shards` to 5 and `number_of_replicas` to 1, you will have 10 shards (5 primary shards and 5 replica shards).

In OpenSearch, the following code creates an index called test with five shards and one replica.

```
PUT test
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1
  }
}
```

Codec settings

Both OpenSearch and Solr use the `best_speed` codec (LZ4 compression algorithm) by default. Both offer `best_compression` (zlib compression algorithm) as an alternative. OpenSearch

also offers the Zstandard compression algorithm (`zstd` and `zstd_no_dict`). Benchmarking for different compression codecs is also available. For more information, see Index codecs in the OpenSearch documentation.

Solr `<codecFactory>` configuration maps to OpenSearch `index.codec` settings; for example:

```
PUT /my_index/_settings
{
      "codec": "best_compression"
}
```

Commit settings

For near real-time search in OpenSearch, you have to set `refresh_interval`. The default is 1 second, which is suitable for most use cases. To improve speed and throughput, especially for batch indexing, we recommend that you increase `refresh_interval` to 30 or 60 seconds.

```
PUT /my_index/_settings
{
  "refresh_interval": "30s",
}
```

Cache and query configuration comparison

OpenSearch uses percentage-based memory allocation instead of entry count, so it provides more adaptive resource management than Solr as the index size changes. OpenSearch also supports tiered caches, where each tier in a multi-tier cache has its own characteristics and performance levels.

The maximum Boolean clause is a static setting in OpenSearch. You set it at node level by using the `indices.query.bool.max_clause_count` setting.

OpenSearch simplifies many configurations by providing sensible defaults while still allowing fine-tuning through its cluster and index settings APIs. OpenSearch supports various cache configurations and types, such as shard request caches and node query caches.

Additionally, in Amazon OpenSearch Service, the Auto-Tune feature uses performance and usage metrics from your OpenSearch cluster to suggest memory-related configuration changes, including queue and cache sizes and Java virtual machine (JVM) settings on your nodes.

Index segments and merging

OpenSearch default settings for merge policies work well for most workload patterns, and explicit merge configurations are rarely needed.

Both Solr and OpenSearch strive to optimize the balance between indexing performance and search efficiency. The Solr `ramBufferSizeMB` and `maxBufferedDocs` settings are handled internally by OpenSearch. OpenSearch manages compound files automatically.

We recommend that you use the OpenSearch default merge policy settings as a starting point instead of copying settings from Solr. In most cases, you won't need to adjust these advanced settings unless you want to tune performance or you have unusual indexing patterns. If tuning is required, as in the following example, see Index settings in the OpenSearch documentation.

```
PUT /my_index/_settings{
  "index": {
    "merge.policy.segments_per_tier": 10,
    "merge.policy.max_merge_at_once": 10,
    "merge.scheduler.max_thread_count": 3,
    "merge.scheduler.max_merge_count": 7,
    "refresh_interval": "1s"
  }
}
```

Step 2. Migrate search functionality

Solr request handlers define how search requests are processed. In OpenSearch, all searches use the `_search` or `_msearch` endpoint. As a best practice, most OpenSearch users prefer the `_search` API for its simplicity and clarity, whereas search templates are typically reserved for complex, reusable queries.

If you're accustomed to using the `/sql` request handler in Solr, you can use SQL syntax and the Piped Processing Language (PPL) for querying in OpenSearch.

OpenSearch supports spell checking, also known as Did-you-mean, and highlighting, during query time. You don't need to explicitly define search components.

Most API responses are limited to JSON format, except for the compact and aligned text (CAT) API.

If you use the Velocity or XSLT response writer in Solr, you must manage it on the application layer in OpenSearch.

When you migrate the Solr `/select` request handler to OpenSearch, you need to transform the XML-based configuration in Solr to a JSON-based search query in OpenSearch. You can also optionally convert the request handler to a search template.

The following example shows how to transform a Solr `/select` handler configuration to OpenSearch. This handler specifies default search parameters, including results per page (`rows`:10) and default search field (`df`: `text`).

```
<!-- Request Handlers -->
  <requestHandler name="/select" class="solr.SearchHandler">
    <lst name="defaults">
      <str name="echoParams">explicit</str>
      <int name="rows">10</int>
      <str name="df">_text_</str>
    </lst>
  </requestHandler>
```

To migrate this handler, you first create the equivalent OpenSearch DSL query that matches the handler functionality. The following example demonstrates a basic search on the `_text` field with a size limit of 10.

```
GET /your_index/_search
{
  "size": 10,
  "query": {
    "match": {
      "_text": "your search term"
    }
  }
}
```

As a best practice, we recommend that you use the OpenSearch `_search` API to search directly. If you need reusable queries, you can create a search template for the equivalent query. The following example shows how to convert the query into a template by using mustache syntax with default parameters. The `rows` parameter in Solr maps to `size` (default: 10) in OpenSearch, and `df` maps to `default_field` (default: `text`). You can render the query by using the `_render/template` API or execute the template by using the `_search/template` API.

```
PUT _scripts/select_handler
{
```

```
  "script": {
    "lang": "mustache",
    "source": {
      "size": "{{#size}}{{size}}{{/size}}{{^size}}10{{/size}}",
      "from": "{{#from}}{{from}}{{/from}}{{^from}}0{{/from}}",
      "query": {
        "query_string": {
          "query": "{{query_string}}",
          "default_field": "{{#df}}{{df}}{{/df}}{{^df}}_text_{{/df}}"
        }
      }
    }
  },
  "params": {
    "size": "Number of results to return (default: 10)",
    "from": "Starting offset for results (default: 0)",
    "query_string": "The search query string (required)",
    "df": "Default field to search (default: _text_)"
  }
}

# To render the query for this template, applications would call:
POST _render/template
{
  "id": "select_handler",
  "params": {
    "query_string": "your search terms"
  }
}
```

Step 3. Migrate document processing pipelines

In Solr, update request processors (URPs) are essential for document transformation during indexing. In OpenSearch, this functionality is provided through ingest pipelines. For the `updateRequestProcessorChain`, OpenSearch provides the ingest pipeline APIs, which let you enrich or transform data before indexing. You can chain multiple processor stages to form a pipeline for data transformation. Processors include Grok, CSV, JSON, KV, Rename, Split, HTML strip, Drop, Script, and others. For more information, see Ingest processors in the OpenSearch documentation.

However, we strongly recommend that you perform data transformations in your extract, transform, load (ETL) layer to remove this processing complexity from OpenSearch. You can

use Amazon OpenSearch Ingestion (OSI), which provides a framework and default processes for data transformation. OSI is built on OpenSearch Data Prepper, which is a server-side data collector that can filter, enrich, transform, normalize, and aggregate data for downstream analytics and visualization.

OpenSearch also provides search pipelines, which are similar to ingest pipelines but tailored for search time operations. Search pipelines make it easier for you to process search queries and search results within OpenSearch. As of OpenSearch version 3.2 , available search processors include filter query, neural query enricher, normalization, rename field, script processor, and personalized search ranking, with more to come.

The following example shows how to transform a Solr `updateRequestProcessorChain` to OpenSearch. This chain includes two processors: a built-in `TrimFieldUpdateProcessorFactory` for whitespace trimming and a custom `CustomPriceTaxProcessorFactory` for price calculations.

```
<!-- updateRequestProcessorChain -->
<updateRequestProcessorChain name="standard">
  <processor class="solr.TrimFieldUpdateProcessorFactory"/>
  <processor class="com.mycompany.CustomPriceTaxProcessorFactory"/>
</updateRequestProcessorChain>
```

The two migration options are Amazon OpenSearch Ingestion (recommended) or using a custom plugin.

Using Amazon OpenSearch Ingestion (recommended)

We recommend that you use Amazon OpenSearch Ingestion, because it provides:

- Better separation of data transformation logic
- Rich set of built-in processors
- More efficient processing outside of OpenSearch
- Easier maintenance and monitoring
- Scalable data processing pipeline

To use this approach, you identify all the processors in the Solr chain and map them to OpenSearch Ingestion processors. Identify any custom processors and analyze the functionality to

determine whether it can be achieved through a combination of OpenSearch processors. You can then set it as a default pipeline for an index.

The following example creates an Amazon OpenSearch Ingestion pipeline with two processors. The first is a trim processor that removes leading and trailing whitespace from the `product_name` field. The second is an AWS Lambda processor that handles price calculations by applying a 10% tax rate to the price field. This is similar to the `CustomPriceTaxProcessorFactory` functionality in the custom Solr processor. We do not recommend using painless scripts for custom implementations, because they can affect performance.

First, create a Lambda function that implements the custom logic for price calculation:

```
def handler(event, context):
    if 'price' in event:
        price = float(event['price'])
        tax_rate = 0.10
        event['price'] = round(price * (1 + tax_rate), 2)
    return event
```

Then create an Amazon OpenSearch Ingestion pipeline:

```
pipeline:
  source:
    file:
      path: "/full/path/to/logs_json.log"
      record_type: "event"
      format: "json"

  processor:
    - trim_string:
        with_keys:
          - "product_name"
    - aws_lambda:
        function_name: "calculateTax"
        invocation_type: "request-response"
        aws:
          region: "us-east-1"

  sink:
    - opensearch:
        hosts: ["https://your-opensearch-domain:443"]
        index: "my_index"
```

Using custom plugins

When you migrate custom plugins from Solr to OpenSearch, we recommend that you first evaluate OpenSearch native features and existing plugins to determine whether they provide the desired functionality before you consider custom development. This approach ensures optimal utilization of OpenSearch native capabilities while maintaining flexibility for custom requirements.

The preceding example follows this approach. It uses an Amazon OpenSearch Ingestion pipeline and implements a custom Lambda processor to migrate `CustomPriceTaxProcessorFactory` by using similar logic.

If native features don't meet your requirements, Amazon OpenSearch Service supports custom plugin development and deployment. For more information, see the Amazon OpenSearch Service documentation.

Sizing your OpenSearch cluster

When you move from Solr to OpenSearch, using the proper sizing is vital for optimal performance and cost management. Start by analyzing your current Solr setup: CPU usage, memory, and performance metrics will serve as your baseline.

You might currently be running an older version of Solr. If so, when you migrate to OpenSearch, you'll typically experience better performance with similar resource allocation, because OpenSearch runs on a newer version of Lucene than your existing Solr deployment. This results in improved search capabilities and optimizations that the updated Lucene engine provides.

Search workloads are usually read-heavy, so we recommend that you prioritize response times by carefully planning replicas, shard sizes, and resource allocation. Use this migration as an opportunity to fix any existing performance issues.

You can start by matching your current Solr resources in OpenSearch. Use the same primary shard size, shard count, CPU, and physical memory for your OpenSearch sizing. Or, you can recalculate the size by using the standard OpenSearch sizing approach, because OpenSearch's optimizations might deliver better performance with the same resources. The goal is to create an improved, efficient search infrastructure instead of replicating your existing setup. For more information, see Sizing Amazon OpenSearch Service domains in the AWS documentation.

For example, consider an ecommerce search platform that implements a product catalog search. Let's say that the search handles 50 million documents and 1000 queries per second (QPS) at peak traffic. The following sections show how the search platform is sized in Solr and OpenSearch.

Solr cluster topology

The following tables specify Solr cluster sizing for the ecommerce search platform example.

Solr component	Value
Total nodes	15
Data nodes	12
ZooKeeper nodes	3
Primary shards	12

Solr component	Value
Replication factor	2
Total shards	24 (12 primary shards and 12 replica shards)

Solr node specification	Value
CPU	4 cores (Intel Xeon 2.4 GHz)
Total RAM	32 GiB
JVM heap	16 GiB
Operating system or file system cache	16 GiB
Disk	100 GiB SSD

Solr data characteristic	Value
Index size (primary)	540 GiB
Index size (total with replica)	1.1 TiB
Shard size	45 GiB
Document count	50 million
Document per shard	Approximately 4.2 million

Solr resource distribution	Per node	Cluster total
CPU cores	4	48 (data nodes)
RAM	32 GiB	192 GiB (data nodes only)

Solr resource distribution	Per node	Cluster total
Storage	100 GiB	1.2 TiB (data nodes only)

To provision identical resources in OpenSearch, keep the following ratios in mind:

- Number of CPUs for every shard
- Amount of JVM for every shard

Equivalent OpenSearch sizing recommendations

The following tables provide OpenSearch sizing recommendations for the Solr clusters in the previous section.

OpenSearch component	Value
Instance type	`r7g.2xlarge`
Data nodes	6
Master nodes	3
Primary shards	12
Total shards	24

OpenSearch node specification	Value
CPU	8 vCPU cores
Total RAM	64 GiB
JVM heap	32 GiB

OpenSearch node specification	Value	
OS or buffer cache	32 GiB	
Disk	200 GiB SSD	

OpenSearch resource distribution	Per node	Data nodes combined
CPU cores	8	48 (data nodes)
RAM	64 GiB	384 GiB (data nodes only)
Storage	200 GiB	1.2 TiB (data nodes only)

Migrating security features

The migration of security assets from Solr to OpenSearch involves extracting existing user credentials and permissions from the `security.json` file in Solr and recreating them in OpenSearch by using the OpenSearch Security API. This process maps Solr authentication and rule-based authorization to the OpenSearch role-based access control (RBAC) system. It ensures that all security policies and user access levels are maintained while taking advantage of the enhanced security features that OpenSearch provides.

The following table compares the security features supported in Solr and OpenSearch.

Feature	Supported in Solr?	Supported in OpenSearch?
Basic authentication	Yes	Yes
JWT authentication	Yes	Yes
Kerberos authentication	Yes	Yes (in self-managed OpenSearch)
Certificate authentication	Yes	Yes (in self-managed OpenSearch)
Hadoop authentication	Yes	No
Amazon Cognito authentication for dashboard access	No	Yes
SAML 2.0 standard authentication for OpenSearch Dashboards	No	Yes
AWS IAM Identity Center authentication and authorization	No	Yes
Native RBAC authorization	Yes (supports external role-based authorization)	Yes (built-in)

Feature	Supported in Solr?	Supported in OpenSearch?
Domain access policy	No	Yes
Encryption at rest	Yes	Yes (default only in managed service)
Encryption at transit	Yes	Yes
Document and field-level security	No	Yes
Multi-tenancy for dashboard	No	Yes

Additional considerations:

- ZooKeeper stores sensitive Solr security information, so protecting ZooKeeper nodes through multiple security measures is critical. This adds complexity to ensure that your Solr cluster remains secure and compliant.

  In contrast, OpenSearch doesn't use file-based storage for its security configuration, which eliminates these security complexities.
- OpenSearch provides two methods for creating security assets: through its API or through the OpenSearch Dashboards user interface. Both approaches offer a straightforward implementation for security configuration management.

To create users and roles from OpenSearch Dashboards, see Defining users and roles in the OpenSearch documentation.

Migrating indexing components

This section explains how to migrate data from Solr to Amazon OpenSearch Service by using a systematic approach that maintains data integrity and search functionality. It covers proven migration strategies, from planning through execution, and focuses on minimizing service disruption and optimizing performance. The information in this section helps you:

- Plan and implement data migration strategies
- Implement efficient migration approaches
- Maintain data integrity and service continuity
- Use AWS managed services effectively

Topics

Understanding data ingestion

In Solr, you can implement extract, transform, load (ETL) workflows by using various approaches that are optimized for different use cases and data sources. These approaches provide proper data extraction, transformation to match your schema, and loading into the search index while maintaining data integrity and search optimization.

The key components of an ETL workflow are:

- Extract: Get data from source systems (databases, file systems, web feeds, and APIs).
- Transform: Process, clean, format, and validate data.
- Load: Index processed data into the target system.

Solr provides request handlers that process incoming requests to add, update, or delete documents in the index. The two primary handlers used for this purpose are the update handler and the Data

Import Handler (DIH). You can use these request handlers to implement ETL processes for ingestion of data into Solr.

Update handler

The update handler is the main endpoint for receiving index updates in Solr. It accepts documents in several formats, including:

- XML: The traditional format for adding and updating documents in the index
- JSON: A modern and widely used format for indexing single documents, a list of documents, or update commands
- CSV: Allows for straightforward ingestion of comma-separated data
- Javabin: An optimized binary format that is used primarily by the SolrJ Java client

The update handler is ideal for direct, programmatic data ingestion where an external application or process pushes data to Solr.

Data Import Handler (DIH)

The Data Import Handler (DIH) is a Solr request handler that pulls data directly into Solr from structured sources such as relational databases, XML files, and RSS feeds by using a configuration-driven approach that's defined in a `data-config.xml` file. DIH supports multiple data sources through Java Database Connectivity (JDBC), file systems, and web feeds; offers both full and incremental (delta) imports for keeping indexes updated; and includes built-in transformers for data modification before indexing.

However, DIH has been deprecated and was removed from Solr 9.0. Solr now recommends alternatives such as using custom ingestion clients with the update handler, community-maintained DIH plugins, or streaming expressions.

Other indexing-related handlers

In addition to the main update handlers, the following tools can help with data ingestion for specific use cases:

- Extracting request handler (Solr Cell): This handler uses Apache Tika to automatically extract text and metadata from rich-text documents such as PDFs and Microsoft Office files.

- Update request processors: These are plugins that can be chained together to pre-process documents before they are indexed. They can perform various tasks, such as language detection, dropping fields, or updating fields.

Common ETL processes in Solr

Organizations typically use the following ETL processes in Solr:

- Database ETL: DIH provides a streamlined ETL process for RDBMS sources by automatically mapping database columns to fields and handling incremental updates. The SolrJ API enables programmatic control for custom ETL workflows when importing from file systems or other structured data sources. You can design a composite ETL pipeline where database records contain metadata and file path references, and use DIH with custom entity processors to fetch metadata and enrich it with file content.
- Web content ETL: Apache Nutch integration delivers a systematic ETL pipeline for web content. Apache Nutch handles crawling and extraction, transforms HTML content into structured data, and loads it directly into the index through native integration (DIH).

Data migration planning

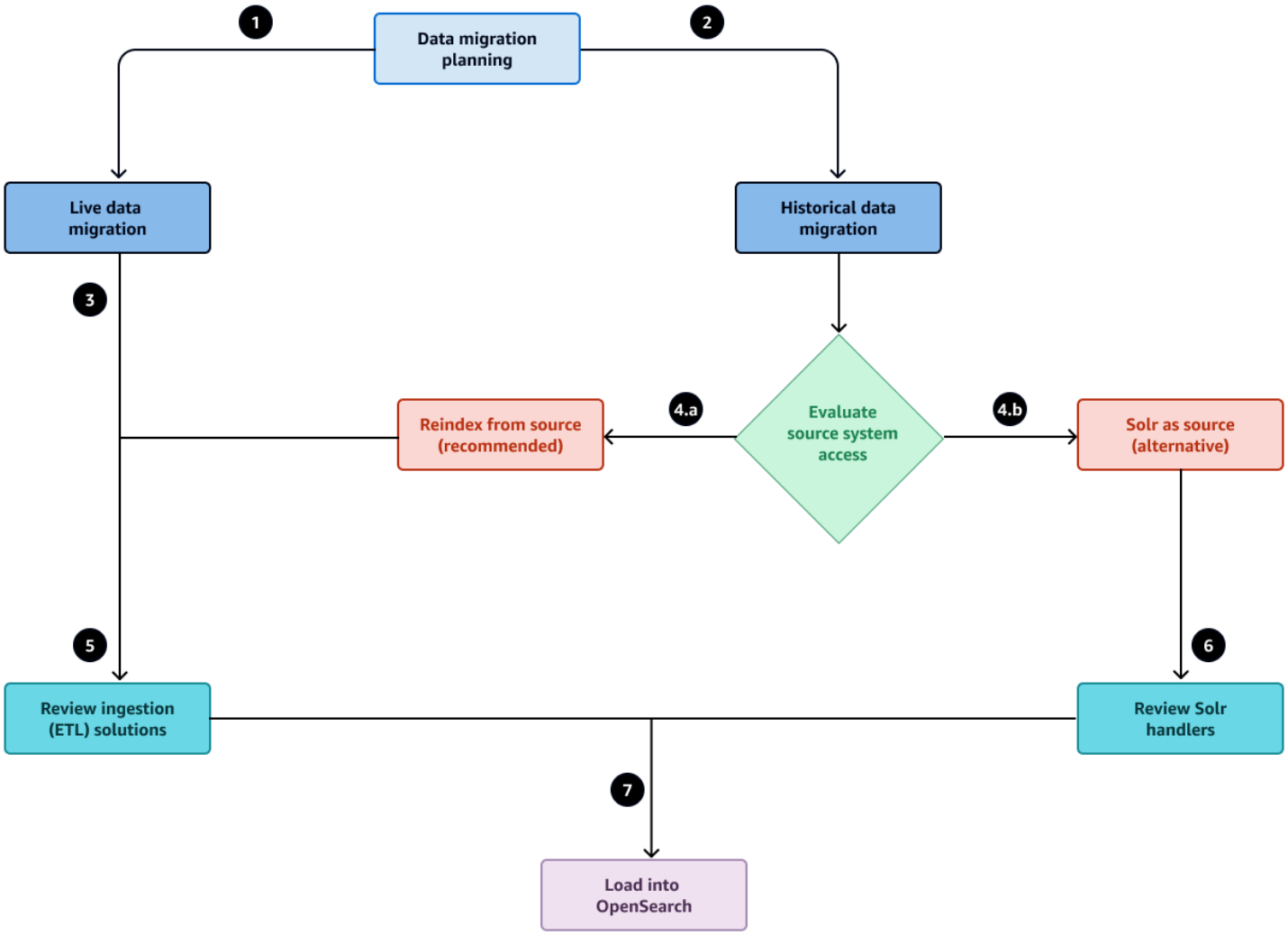
This section explores data migration paths and helps you select the most appropriate strategy for your use case. Before you begin your data migration to OpenSearch, choose the approach that best fits your requirements.

Assess your current environment

When you decide to migrate from Solr to OpenSearch, you start a process that requires careful planning and strategic decision-making. The first crucial step is to assess your current environment and determine the most effective path forward.

The initial evaluation should focus on your transformation requirements. If you have straightforward transformations and robust ETL processes in Solr, adapting the current architecture with OpenSearch modifications might be optimal. On the other hand, if you need advanced capabilities or modernization, consider using Amazon OpenSearch Ingestion (OSI).

The following diagram illustrates the two major approaches to data migration, which run in parallel.



1. **Live data migration** follows a path from 1, to 3, 5, and 7 in the diagram. It captures real-time change data from various sources and ingests it into OpenSearch through ETL solutions.
2. **Historical data migration** follows a path from 2 to 4-5-7 or 6-7. It migrates past data to OpenSearch and provides two options after evaluating source system access:
   - **4.a Reindex from source (recommended)**. When you migrate to Amazon OpenSearch Service, building from your source systems provides the most reliable path. This approach gives you direct access to your authoritative data sources, whether they're in Amazon Relational Database Service (Amazon RDS), Amazon DocumentDB, Amazon Simple Storage Service (Amazon S3), or other storage systems. You maintain complete control over data quality and can optimize your schema design during migration. The source system approach supports formats such as JSON, CSV, and Apache Parquet, so you can work with your existing data structures while implementing the necessary transformations. The next step after you choose this migration option is to review existing ingestion (ETL) solutions (5).

- **4.b Solr as source**. Consider direct Solr migration when your source systems are inaccessible or when you need a faster migration path. This approach requires a stable Solr deployment with sufficient resources to handle the additional read load during migration. You can use either the Solr select handler with the `cursorMark` parameter for reliable pagination, or the export handler for optimized streaming. Although this method offers quicker implementation, it requires careful monitoring of your Solr cluster's performance and network bandwidth to ensure successful data transfer. This approach focuses solely on migrating historical data, so any tools or processes developed for this migration should be considered temporary and disposable. Continuous ingestion and ongoing data updates from the source system will need a separate, permanent solution after the historical data migration is complete. The next step after you choose this migration option is to review Solr handler solutions (6).

The recommended approach is to reindex from the original source system. Consider using Solr as a source as an alternative when direct source access is limited.

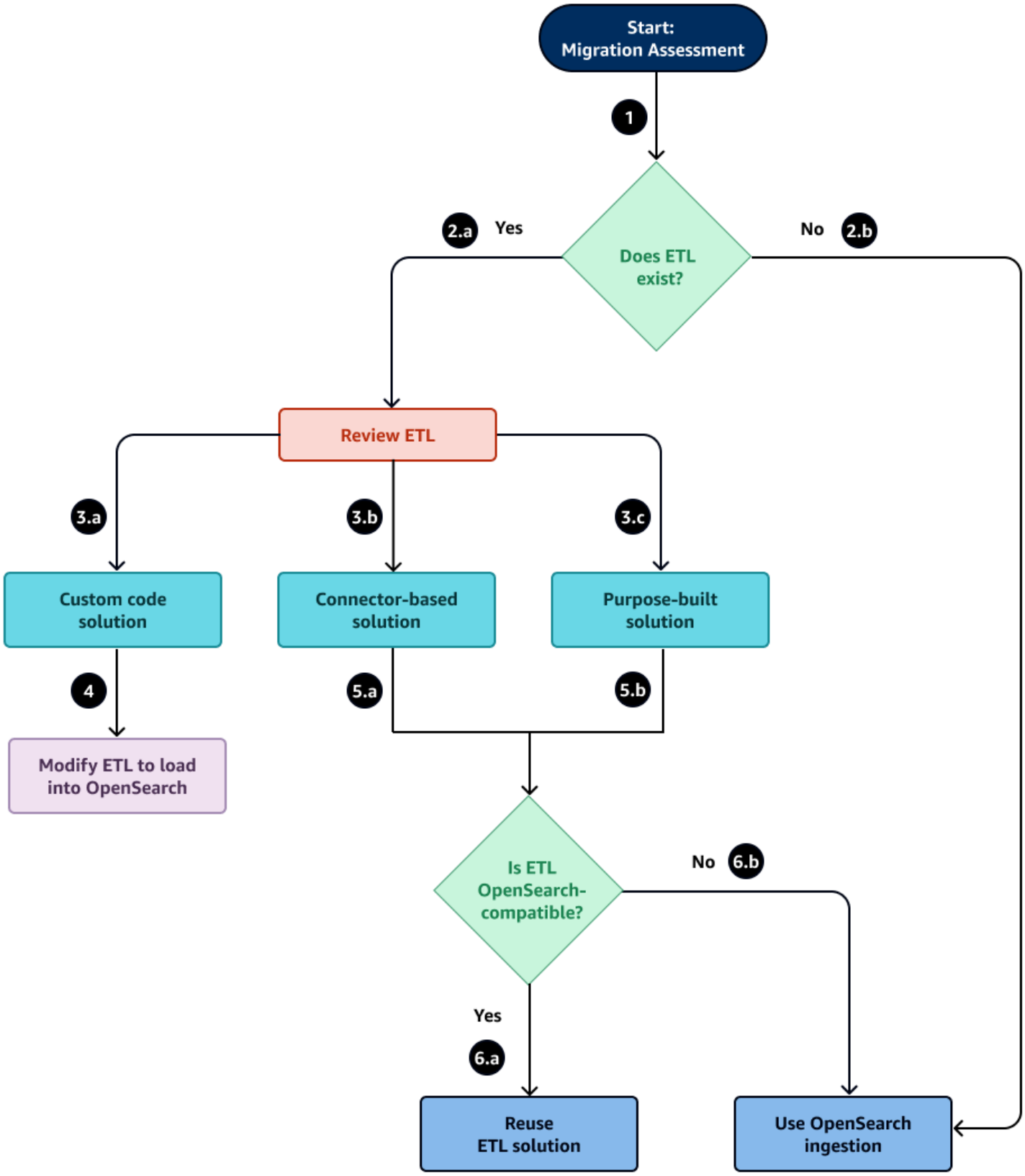
ETL assessment

After you select your migration approach, you can perform a detailed ETL assessment. This assessment helps determine specific implementation requirements based on your chosen path and existing infrastructure.

- **Reviewing ingestion (ETL) solutions.** If your organization already has an established ETL pipeline, evaluate its effectiveness and compatibility with OpenSearch. Identify which category your existing ETL solution falls into:
  - Custom application development (for example, your own SolrJ-based solution)
  - Connector-based solution (for example, using Solr DIH, request handlers, or Apache Nutch)
  - Purpose-built integration (for example, Apache Tika connectors)
- **Reviewing Solr handlers.** Solr provides handlers such as select and export by default. These help export millions of records.

Data migration process

The following diagram provides a decision matrix for your migration assessment.



The following sections describe the process illustrated in the diagram in more detail.

Determine whether an ETL solution exists

This decision point determines whether you have an existing ETL solution in Solr, leading to two paths:

- If you have an existing ETL solution (2.a in the diagram), follow the instructions in the next section to adapt your implementation type to work with OpenSearch.
- If you don't have an existing ETL solution (2.b in the diagram), you have an opportunity to build a modern, cloud-native data pipeline by using OSI.

Evaluate your current ETL

The ETL pathway consists of:

- Light transformation: Adapt existing ETL by incorporating an OpenSearch sink (the target for your data).
- Heavy transformation: Preserve current ETL infrastructure while transitioning from SolrJ to the OpenSearch bulk API.

When you migrate data from your current data sources, you typically have ETL solutions that align with one of three categories:

- Custom code solutions (3.a in the previous diagram)
- Connector-based solutions (3.b in the diagram)
- Purpose-built solutions (3.c in the diagram)

Custom code solutions (high complexity)

If you've built custom data ingestion solutions for Solr, migrating to Amazon OpenSearch Service requires a systematic approach that preserves your existing functionality while leveraging AWS Cloud capabilities. This section guides you through the migration process for your custom applications.

1. **Understand your current environment**. Begin by examining your existing custom implementation. Your application likely contains specific data processing logic, unique ingestion patterns, and custom post-processing requirements that you'll need to maintain after migration.

Start with a comprehensive review of your current system, focusing on how your application interacts with Solr and identifying critical functionality that must be preserved.

2. **Plan your migration**. Before you modify any code, conduct a thorough assessment across three key areas:
   - Analyze your codebase to understand its scope and complexity. Document the programming languages in use, map out your data transformation logic, and review your current ingestion patterns. This analysis helps identify potential challenges and opportunities for optimization.
   - Evaluate your architecture. Whether your application runs on premises or on Amazon Elastic Compute Cloud (Amazon EC2), consider how you can modernize your deployment model. This might involve adopting containerization, implementing serverless components, or using other AWS managed services.
   - Examine your data processing workflows. Understanding how your application transforms and loads data helps ensure a smooth transition to Amazon OpenSearch Service while maintaining data integrity.
3. **Implement changes**. Focus on two primary areas:
   - Update your code to work with Amazon OpenSearch Service. This involves:
     - Modifying endpoint configurations to connect to your OpenSearch domain.
     - Updating client libraries to use the OpenSearch SDK.
     - Implementing AWS authentication mechanisms.
     - Testing data synchronization to ensure consistency.
   - Consider modernization opportunities, such as:
     - Evaluating OSI for simplified data loading.
     - Exploring AWS managed services that could replace custom components.
     - Planning a gradual transition from self-managed solutions to reduce risk.

The following sample code shows how to migrate document indexing code from an Apache SolrJ client to an OpenSearch client.

JSON format:

```
{
    "id": "123",
    "title": "Sample Document",
    "content": "This is the document content"
```

```
}
```

Indexing in a SolrJ client:

```
//Using SolrInputDocument
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "123");
doc.addField("title", "Sample Document");
doc.addField("content", "This is the document content");
solrClient.add(doc);

// Commit the changes
solrClient.commit();
```

Indexing in an Opensearch client:

```
// Using Map
Map<String, Object> document = new HashMap<>);
document.put("title", "Sample Document");
document.put("content", "This is the document content");
IndexRequest request = new IndexRequest("index_name").id("123").source(document);

// Run the index request
IndexResponse response = client.index(request, RequestOptions.DEFAULT);
```

Connector-based solutions (medium complexity)

If your Solr implementation relies on prebuilt connectors for data ingestion, migrating to Amazon OpenSearch Service requires careful evaluation of your existing integrations. This section guides you through migrating your connector-based solutions while maintaining their functionality.

1. **Understand your current connectors**. Connectors serve as crucial bridges between your data sources and search infrastructure. In Solr environments, you might be using Database Import Handler (DIH) for database integration, XML or JSON handlers for file processing, Apache Tika for document parsing, Apache Nutch for web content crawling, or Fluent Bit or Logstash for real-time data processing. These connectors provide standardized ways to ingest and transform data while reducing development overhead.
2. **Plan your migration**. Start by assessing your current connector ecosystem. Examine how each connector interfaces with your data sources and what transformations they perform. For

example, if you're using DIH to import database records, document the mapping configurations and any custom transformations you've implemented.

Based on your assessment, you have three main implementation options:

- Adapt compatible connectors. If your current connectors support Amazon OpenSearch Service, you can modify their configurations to point to your new OpenSearch domain. This approach minimizes changes to your existing architecture while leveraging familiar tools.
- Implement OSI. For scenarios where direct compatibility isn't possible, OSI provides a managed alternative. This service handles data ingestion with built-in support for various data sources and formats.
- Use AWS managed services. Consider AWS Cloud solutions that can replace your current connectors. For example, you can consider using AWS Lambda for your ingestion logic instead of using AWS Database Migration Service (AWS DMS). If you want to stream data to Amazon OpenSearch Service without staging data, consider using Amazon Data Firehose for streaming data.

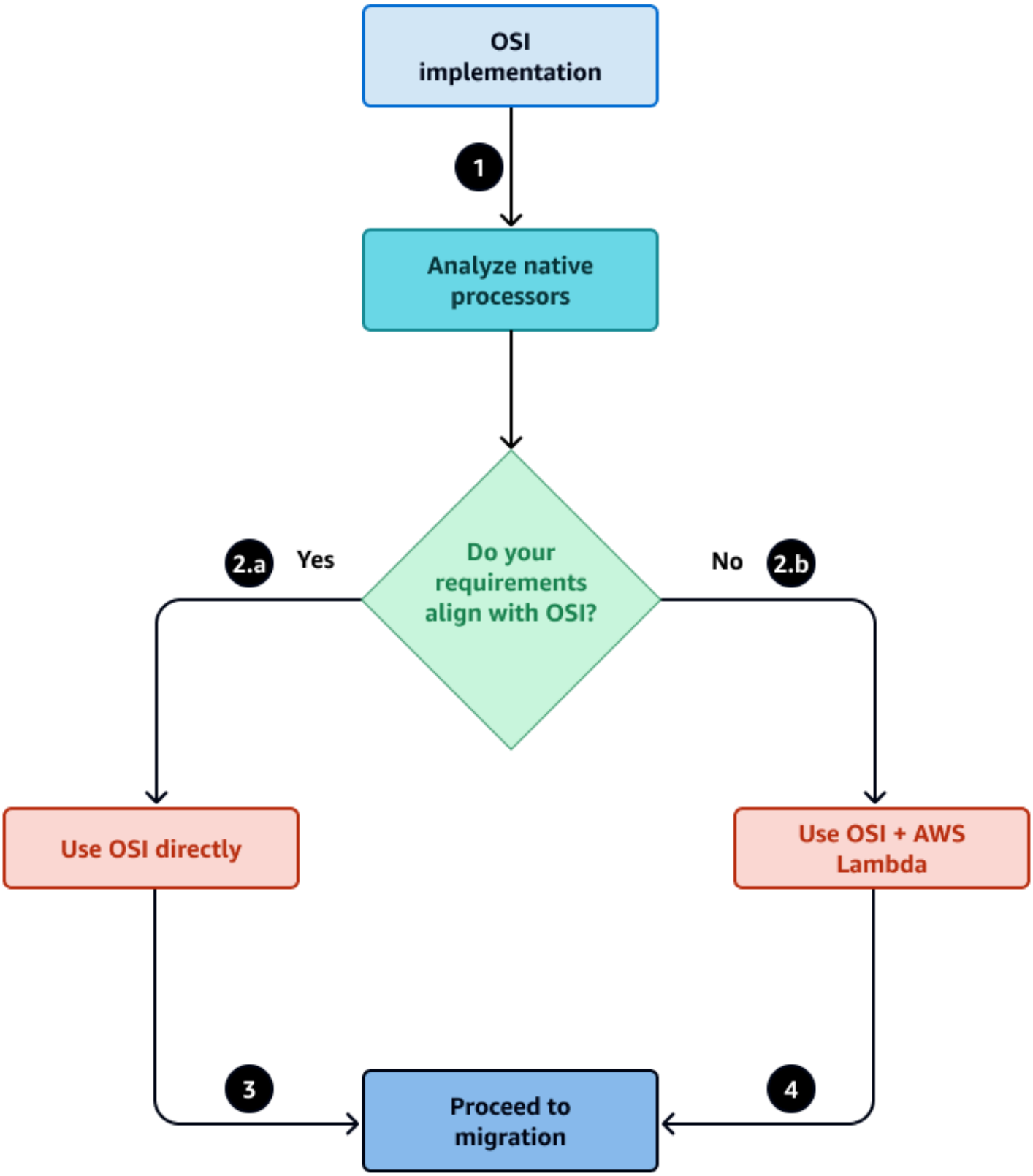
3. **Implement changes**. Many Solr connectors work with XML formats, whereas OpenSearch prefers JSON. Plan your transformation approach.

Purpose-built integrations

When your Solr implementation uses purpose-built tools for data ingestion, migrating to Amazon OpenSearch Service often presents the most straightforward path.

Purpose-built tools simplify your migration journey by providing streamlined migration paths, simple configuration updates, minimal code modifications, and ready-to-use integration patterns.

OSI provides a managed service for migrating your data to Amazon OpenSearch Service. This section guides you through the implementation process, from initial assessment to production deployment, as illustrated in the following diagram.


OSI implementation
1
Analyze native processors
Do your requirements align with OSI?
2.a
Yes
No
2.b
Use OSI directly
Use OSI + AWS Lambda
3
4
Proceed to migration

1. **Initial assessment**. Begin your implementation journey by evaluating your migration requirements. Consider your current data volumes, throughput needs, and transformation requirements. During this phase, analyze:
   - Your application's performance requirements
   - Data source compatibility with OSI
   - Service quotas and limitations
   - Resource requirements
2. **Processor evaluation**. Based on your assessment, determine your processing needs:
   - **2.a. Native processors**. When your requirements align with OSI's built-in capabilities, you can use a native processor. An example configuration includes:

     ```
     processor:
     - type: date
     field: timestamp
     formats: ["yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"]
     ```

     Implementation steps include:
     - Configuring data source connections
     - Setting up transformation pipelines
     - Implementing monitoring
     - Testing performance
   - **2.b. Custom transformations**. For complex transformations that require additional processing, you can use AWS Lambda. For example:

     ```
     def process_record(event, context):
     # Custom transformation logic
     return transformed_data
     ```

     Key considerations:
     - Lambda function implementation
     - Custom transformation logic
     - Error handling setup
     - Performance monitoring

3. **Direct implementation path**. For implementations that use native processors, you can configure pipelines directly. For example:

```
pipeline:
source:
type: s3
bucket: your-bucket sink:
type: opensearch
domain: your-domain
```

4. **Enhanced implementation path**. For implementations that include Lambda processors, you can configure pipelines as follows:

```
pipeline:
source:
type: s3 processor:
- type: lambda
function_arn: your-lambda-arn
sink:
type: opensearch
```

Your migration approach depends on your tool's compatibility with Amazon OpenSearch Service.

- Compatible tools. When your tools support Amazon OpenSearch Service, focus on configuration updates. Implementation steps include:
  - Updating endpoint configurations.
  - Modifying authentication settings.
  - Testing performance and reliability.
  - Monitoring data consistency.
- Incompatible tools. For tools that lack OpenSearch compatibility:
  - Assess transformation requirements.
  - Evaluate data flow patterns.
  - Document integration points.

Based on your compatibility assessment, choose one path:

- For compatible solutions:
  - Reuse existing tools and update OpenSearch as the load endpoint.
  - Update configurations.

- Test performance.
- Monitor operations.

- For a new implementation with OSI, set up a pipeline; for example:

```
# Example: OSI pipeline setup
version: 1
pipeline:
source:
type: s3
sink:
type: opensearch
```

OSI provides zero-ETL to the following supported data sources:

- Amazon S3 for bulk data
- Amazon DynamoDB for NoSQL data
- Amazon DocumentDB for document data
- Amazon Aurora for SQL data
- Apache Kafka for streaming data

The typical migration process starts from Solr, exports the data to JSON, places it in an S3 bucket, ingests it into OSI, and moves it to OpenSearch. By following this structured approach, you can implement OSI effectively while maintaining AWS best practices for performance, reliability, and operational excellence.

So far, this section focused on how configurations can be rewired. The next section explains how data can be migrated.

Data loading strategies

Initial data migration

The initial data migration represents the most critical phase of your Amazon OpenSearch Service migration. It requires careful orchestration to ensure data integrity while minimizing service disruption. The initial data migration consists of four phases.

Phase 1: Pre-migration preparation. This phase involves coordination with application teams to schedule maintenance windows and implement write operation controls. Tasks include:

- Implementing application-level read-only modes to prevent new data modifications.
- Ensuring that all pending transactions are completed before proceeding.
- Documenting the timestamp when write operations are suspended.
- Communicating status to all stakeholders throughout the process.

Phase 2: Data transfer. In this phase, you run your chosen extraction method while maintaining comprehensive monitoring. Tasks include:

- Implementing progress tracking and logging for visibility into transfer status.
- Configuring checkpointing mechanisms to enable recovery from interruptions.
- Monitoring AWS service quotas and requesting increases if necessary.
- Maintaining detailed audit logs for compliance and troubleshooting purposes.

Phase 3: Data validation and verification. This phase includes both quantitative and functional validation.

Quantitative validation:

- Compare document counts between source Solr and destination Amazon OpenSearch Service.
- Validate field mappings and data types across representative document samples.
- Execute checksum or hash comparisons for critical data fields.
- Verify index statistics and storage utilization metrics.

Functional validation:

- Run representative search queries against both systems.
- Compare search result relevance and ranking.
- Validate faceting, aggregation, and filtering functionality.
- Test application-specific search features and use cases.

Phase 4: Application cutover. This phase requires updating your application configurations to use Amazon OpenSearch Service:

- Update connection strings and endpoint configurations.

- Modify query syntax for Amazon OpenSearch Service compatibility.
- Update authentication and authorization configurations for AWS Identity and Access Management (IAM).
- Implement gradual traffic shifting where possible to minimize risk.

Before you cut over, make sure that comprehensive monitoring mechanisms are in place:

- Configure Amazon CloudWatch dashboards for key performance indicators.
- Set up CloudWatch alarms for critical metrics.
- Implement AWS X-Ray tracing for application performance monitoring.
- Configure AWS CloudTrail logging for audit and compliance requirements.

Incremental data synchronization

For scenarios where data modifications occurred during migration, implement change data capture (CDC) mechanisms by using AWS Database Migration Service (AWS DMS) or Amazon Kinesis Data Streams.

- For relational database sources, use AWS DMS to capture and replicate ongoing changes:
  - Configure AWS DMS tasks for continuous replication.
  - Implement transformation rules for Amazon OpenSearch Service compatibility.
  - Monitor replication lag and performance metrics.
  - Handle data type conversions and schema differences.
- For high-velocity data sources, use Amazon Kinesis Data Streams integration to implement real-time change streaming.

Best practices and recommendations

- For fast data ingestion, disable replica shards to increase the indexing rate. After data is ingested, you can enable the desired number of shards.
- Implement least-privilege access principles for all migration components.
- For data encryption:
  - Configure encryption for data in transit and at rest.

- Enable Amazon OpenSearch Service domain encryption at rest by using AWS Key Management Service (AWS KMS).
- Configure node-to-node encryption for inter-cluster communication.
- Use SSL/TLS for all client connections to Amazon OpenSearch Service.
- Encrypt data that's stored in Amazon S3 during the migration process.

Migrating search queries

The previous sections discussed migrating the schema and other necessary configurations and historical data from Solr to OpenSearch. Now it's time to translate queries to OpenSearch and test them. This section discusses how to migrate search queries into OpenSearch and best practices for this migration.

Both Solr and OpenSearch use Apache Lucene to perform search. However, there are some differences in how Solr and OpenSearch process queries.

Solr query processing consists of this flow:

Client request → request handler → query parser → query execution → response writer

OpenSearch query processing involves this flow:

Client request → REST API → query DSL parser → query execution → JSON response

As discussed earlier in this guide, Solr request handlers receive requests and query parsers interpret the query string (q parameter) and convert it into Lucene `Query` objects. Different parsers support different syntax and features. Here is an example of a search handler, which is a specific type of request handler:

```
<requestHandler name="/select" class="Solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">20</int>
    <str name="df">content</str>
    <str name="defType">edismax</str>
  </lst>
</requestHandler>
```

The following Solr query uses this search handler:

```
# With filters
http://localhost:8983/solr/books/select?q=*:*&fq=category:programming
```

This Solr query specifies a query parser:

```
http://localhost:8983/solr/books/select?q=title:java&defType=lucene
```

Converting Solr query parameters to OpenSearch DSL

As an example, let's use the following Solr query:

```
http://localhost:8983/solr/books/select?q=*:*&fq=category:programming
```

The equivalent OpenSearch query is:

```
GET http://localhost:9200/books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_all": {}
        }
      ],
      "filter": [
        {
          "term": {
            "category": "programming"
          }
        }
      ]
    }
  }
}
```

Key differences

The following table lists the key differences between Solr and OpenSearch search features.

Feature	Solr	OpenSearch
Full-text search	Query parsers: Standard, DisMax, Extended DisMax (eDisMax), Simple	Match, multi-match, query string, match phrase, and so on
Faceting	`facet`, `facet.pivot` , `json.facet` parameters	Metric, bucket, and pipeline aggregations

Feature	Solr	OpenSearch
Filtering	Filter query (`fq`) parameter	Filter context
Boosting	DisMax, eDisMax query	Boosting, disjunction, function score
Highlighting	`unified`, `original`, `fastVector` parameters	`unified`, `fvh`, `plain`, `semantic` highlighters
Suggestions	Suggester component	Autocomplete
Geospatial	Spatial search	Geographic and XY queries

The following table lists frequently used query types in Solr and how to rewrite them in OpenSearch.

Query type	Solr	OpenSearch

Basic Boolean query	**Solr:**
	`# Simple field query`
	`q=title:java`
	
	`# Multiple field queries`
	`q=title:java AND author:sm ith`
	`q=title:java OR author:py thon`
		`// Simple field query`
		`{`
		`  "query": {`
		`  "term": {`
		`  "title": "java"`
		`  }`
		`  }`
		`}`
		
		`// Multiple field queries - AND`
		`{`
		`  "query": {`
		`  "bool": {`
		`  "must": [`
		`  {"term": {"title": "java"}},`
		`  {"term": {"author": "smith"}}`
		`  ]`
		`  }`
		`  }`

Query type	Solr	OpenSearch

		`}`
		
		`// Multiple field queries - OR`
		`{`
		`  "query": {`
		`  "bool": {`
		`  "should": [`
		`  {"term": {"title": "java"}},`
		`  {"term": {"author": "python"}}`
		`  ]`
		`  }`
		`  }`
		`}`

Query type	Solr	OpenSearch

Full-text search queries with boost	`# Solr DisMax`
	`q=java programming&defTyp`
	`e=dismax&qf=title^2 content`
	` author`
	
	`# Solr eDisMax with phrase`
	` boost`
	`q=java programming&defTyp`
	`e=edismax&qf=title^3`
	` content^1&pf=title^10&mm=75`
	`%`
		`// Basic multi-match`
		`{`
		`  "query": {`
		`    "multi_match": {`
		`      "query": "java programmi`
		`ng",`
		`      "fields": ["title^2",`
		` "content", "author"]`
		`    }`
		`  }`
		`}`
		
		`// Advanced multi-match with`
		` phrase boost`
		`{`
		`  "query": {`
		`    "bool": {`
		`      "must": [`
		`        {`
		`          "multi_match": {`
		`            "query": "java`
		` programming",`
		`            "fields":`
		` ["title^3", "content"],`
		`            "minimum_`
		`should_match": "75%"`
		`          }`
		`        }`
		`      ],`
		`      "should": [`
		`        {`
		`          "match_phrase": {`
		`            "title": {`
		`              "query": "java`
		` programming",`
		`              "boost": 10`
		`            }`
		`          }`
		`        }`
		`      ]`
		`    }`

Query type	Solr	OpenSearch
		} }

Range queries	# Numeric range
	q=price:[10 TO 50]
	
	# Date range
	q=publishDate:[2020-0
	1-01T00:00:00Z TO 2023-12-3
	1T23:59:59Z]
	
	# Open-ended range
	q=price:[10 TO *]
	q=price:[* TO 50]
		// Numeric range
		{
		"query": {
		"range": {
		"price": {
		"gte": 10,
		"lte": 50
		}
		}
		}
		}
		
		// Date range
		{
		"query": {
		"range": {
		"publishDate": {
		"gte": "2020-01-
		01T00:00:00Z",
		"lte": "2023-12-
		31T23:59:59Z"
		}
		}
		}
		}
		
		// Open-ended range
		{
		"query": {
		"range": {
		"price": {
		"gte": 10
		}
		}
		}
		}

Query type	Solr	OpenSearch

Wildcard and fuzzy queries	`# Wildcard queries`
	`q=title:jav*`
	`q=title:*ava`
	`q=title:j?va`
	
	`# Fuzzy queries`
	`q=title:java~2`
	`q=title:"java programming"~3`
		`// Wildcard queries`
		`{`
		`  "query": {`
		`    "wildcard": {`
		`      "title": "jav*"`
		`    }`
		`  }`
		`}`
		
		`{`
		`  "query": {`
		`    "wildcard": {`
		`      "title": "*ava"`
		`    }`
		`  }`
		`}`
		
		`// Fuzzy queries`
		`{`
		`  "query": {`
		`    "fuzzy": {`
		`      "title": {`
		`        "value": "java",`
		`        "fuzziness": 2`
		`      }`
		`    }`
		`  }`
		`}`
		
		`// Fuzzy phrase query`
		`{`
		`  "query": {`
		`    "match_phrase": {`
		`      "title": {`
		`        "query": "java`
		` programming",`
		`        "slop": 3`
		`      }`
		`    }`
		` }

Query type	Solr	OpenSearch
		`}`
Filter queries	`# Filter queries q=java&fq=category: programming&fq=inStock:true &fq=price:[10 TO 50]`	`{ "query": { "bool": { "must": [ {"match": {"_all": "java"}} ], "filter": [ {"term": {"category": "programming"}}, {"term": {"inStock": true}}, {"range": {"price": {"gte": 10, "lte": 50}}} ] } } }`

Query type	Solr	OpenSearch

Phrase and proximity queries	`# Exact phrase`
	`q="java programming"`
	
	`# Proximity search`
	`q="java programming"~5`
	
	`# Phrase with field`
	`q=title:"machine learning"~2`
		`// Exact phrase`
		`{`
		`  "query": {`
		`    "match_phrase": {`
		`      "_all": "java programming"`
		`    }`
		`  }`
		`}`
		
		`// Proximity search`
		`{`
		`  "query": {`
		`    "match_phrase": {`
		`      "_all": {`
		`        "query": "java programming",`
		`        "slop": 5`
		`      }`
		`    }`
		`  }`
		`}`
		
		`// Phrase with specific field`
		`{`
		`  "query": {`
		`    "match_phrase": {`
		`      "title": {`
		`        "query": "machine learning",`
		`        "slop": 2`
		`      }`
		`    }`
		`  }`
		`}`
		
		You can also use span and interval queries for proximity search.

Query type	Solr	OpenSearch
Aggregation	`# Basic faceting q=*:*&facet=true&facet.field=category&facet.field=author`	`// Basic aggregations { "query": {"match_all": {}}, "aggs": { "categories": { "terms": { "field": "category" } }, "authors": { "terms": { "field": "author" } } } }`

Query type	Solr	OpenSearch
Nested faceting or aggregation	`# nested faceting { "query": "*:*", "facet": { "categories": { "type": "terms", "field": "category", "limit": 10, "facet": { "brands": { "type": "terms", "field": "brand", "limit": 5, "facet": { "avg_price": { "type": "avg", "field": "price" }, "max_price": { "type": "max", "field": "price" } } } } } } }`	`# OpenSearch nested aggregation { "query": { "match_all": {} }, "size": 0, "aggs": { "categories": { "terms": { "field": "category", "size": 10 }, "aggs": { "brands": { "terms": { "field": "brand", "size": 5 }, "aggs": { "avg_price": { "avg": { "field": "price" } }, "max_price": { "max": { "field": "price" } } } } } } } }`

Query type	Solr	OpenSearch

Geospatial queries	`# Geo distance`
	`q={!geofilt pt=37.775`
	`2,-122.4232 sfield=location`
	` d=10}`
		`// Geo distance`
		`{`
		`  "query": {`
		`  "geo_distance": {`
		`  "distance": "10km",`
		`  "location": {`
		`  "lat": 37.7752,`
		`  "lon": -122.4232`
		`  }`
		`  }`
		`  }`
		`}`

The following sections discuss some of the Solr search features and their equivalents in OpenSearch in more detail. After migration to OpenSearch, make sure to test all your queries in OpenSearch and compare the results with your Solr-based system. For your search applications, OpenSearch provides both high-level and low-level clients for multiple languages. For more information, see OpenSearch language clients in the OpenSearch documentation.

Search features

The following sections discuss some of the Solr search features and their equivalents in OpenSearch in more detail. After migration to OpenSearch, make sure to test all your queries in OpenSearch and compare the results with your Solr-based system. For your search applications, OpenSearch provides both high-level and low-level clients for multiple languages. For more information, see OpenSearch language clients in the OpenSearch documentation.

Key conversions and challenges

Search feature conversion is the most complex aspect of migration. It requires the translation of the Solr `SolrFeature` class to OpenSearch DSL templates, the Solr `FieldValueFeature` class to the OpenSearch `field_value_factor` function, and function queries to OpenSearch scripts. Tree models have to be restructured into RankLib XML format, and neural networks require external service implementation. Query syntax conversion from Lucene to JSON DSL and performance optimization strategies differ significantly between the two systems.

Although the core LTR concepts remain the same, the implementation details differ significantly. The migration requires careful conversion of features, models, and queries, but can be largely automated with proper tooling.

Join queries

Solr lets you run join queries to perform inner joins on different datasets to create a normalized dataset.

In OpenSearch, you can use join operations through both Piped Processing Language (PPL) and SQL interfaces to combine data from multiple datasets.

PPL provides a simple join command with a straightforward syntax:

```
source=customer
| join ON c_custkey = o_custkey orders
| head 10
```

SQL offers more granular join control with support for `INNER`, `LEFT OUTER`, and `CROSS` joins; for example:

```
SELECT
    A.Body,
    B.Timestamp
FROM
    <tableNameA/logGroupA> AS A
INNER JOIN
    <tableNameB/logGroupB> AS B
    ON A. 'requestId' = B. 'requestId'
```

Highlighting

Solr highlighting features are quite similar to OpenSearch. They both support the original, fast vector, and unified highlighters, which makes the migration straighforward. Solr supports a few additional parameters such as `fragAlignRatio`, `fragsizeIsMinimum`, `alternateField`, and `fragmenter`. You might need some workarounds for these in OpenSearch. OpenSearch also supports semantic highlighting, which Solr doesn't offer.

Streaming expressions

In Solr, you can use streaming expressions to perform real-time analytics and complex data transformations directly within Solr, without needing to export data to another system for processing. You can use these functions to perform mathematical and statistical operations, aggregations, and additional operations on search results as they are streamed back to the client.

Streaming expressions aren't natively available in OpenSearch. To perform streaming you can use either the `scroll` or `search_after` deep pagination technique to stream data out of OpenSearch.

SQL queries

Solr uses the `/sql` request handler with the Apache Calcite SQL engine and supports both JDBC driver connections and HTTP interfaces, whereas OpenSearch implements SQL through the `_plugins/_sql` REST API endpoint. For example:

```
POST _plugins/_sql
{
  "query": "SELECT * FROM my-index LIMIT 50"
}
```

In OpenSearch, you can implement functionality that's similar to the `/sql` handler, including complex operations such as lookup, join, and subsearch, by using PPL query language commands that are powered by OpenSearch-Calcite integration.

The SQL syntax is largely compatible between systems, but you'll need to update any Solr-specific features such as the `/export` handler for unlimited queries to equivalent mechanisms in OpenSearch, and ensure that field mappings align with the OpenSearch document structure instead of the schema-based approach that Solr uses. For more information about the SQL features in OpenSearch, see SQL in the OpenSearch documentation.

Migrating your queries requires:

- If you're using JDBC, modifying the connection strings from the `jdbc:Solr://zkHost?collection=name` format to the REST-based approach in OpenSearch.
- Adapting query parameters. Solr supports parameters such as `aggregationMode` and `numWorkers` for MapReduce operations, whereas OpenSearch focuses on format specifications such as `format=json/csv/jdbc`.

PPL queries

PPL is a sequential, step-by-step query language that uses the pipe (`|`) operator to combine commands for processing data. It also supports advanced query options such as `join`, `lookup`, and `dedup` (data deduplication). When you migrate SQL handlers from Solr to OpenSearch, you can use PPL to deal with log analysis, data monitoring, or semi-structured datasets, because it offers a more intuitive and readable syntax for sequential data processing compared with traditional SQL queries. For more information, see PPL in the OpenSearch documentation.

Learning to Rank

Learning to Rank (LTR) is a machine learning approach that uses trained models to improve search result ranking in Solr and OpenSearch.

Solr LTR uses Java-based feature classes with `rq` parameter integration and schema-based storage, whereas OpenSearch LTR uses mustache templates with rescore queries and index-based `.ltrstore` storage. Both LTR implementations support linear and tree models.

To migrate LTR functionality from Solr to OpenSearch:

- Migrate LTR models (migrate linear maps directly and convert trees to RankLib XML).
- Update query patterns from `rq={!ltr}` to `rescore.sltr`.
- Validate feature values and model scores for consistency.

For more information about LTR support in OpenSearch, see Learning to Rank in the OpenSearch documentation.

Query debugging

Profile API

For Solr users who migrate to OpenSearch, the Profile API replaces the `debug=timing` parameter that Solr uses for query performance analysis. You can add `"profile": true` to your OpenSearch search requests to get detailed execution breakdowns that are similar to the debug output in Solr.

OpenSearch profiles provide nanosecond-level breakdown for each query component. Profiles are equivalent to query explanations in Solr but provide higher granularity. The OpenSearch response

structure shows breakdowns for query parsing, execution, aggregations, and document retrieval phases, similar to how Solr breaks down query processing time.

Here's an example request that uses the OpenSearch Profile API:

```
GET /testindex/_search?human=true
{
  "profile": true,
  "query" : {
    "match" : { "title" : "rain" }
  }
}
```

For more information, see Profile API in the OpenSearch documentation.

Explain API

You might be using `debug=result` parameter in Solr to understand why a particular document ranks higher or lower in search results. For similar functionality in OpenSearch, you can use the Explain API, which shows a detailed calculation of how the relevance score was calculated for each document. For example:

```
POST opensearch_dashboards_sample_data_ecommerce/_explain/EVz1Q3sBgg5eWQP6RSte
{
  "query": {
    "match": {
      "customer_first_name": "Mary"
    }
  }
}
```

For more information, see Explain API in the OpenSearch documentation.

Migrating the Admin Console

The Solr Admin UI is a web-based interface that users access at `http://hostname:8983/Solr/` to get comprehensive access to Solr configuration and features.

> **Note**
>
> 8983 is the default Admin Console port for Solr, but this port number might vary in non-default configurations.

You use the Solr Admin UI for administration purposes; for example, to monitor clusters, administer collections, design schemas, run Solr queries, and manage the security of your cluster. If you use Solr for analytics, you might be taking advantage of UI integrations such as Apache Zeppelin, Apache Superset, and Grafana for dashboarding.

When you migrate to OpenSearch, you can use OpenSearch Dashboards for both administrative purposes such as managing your schema, security, and roles, and for data visualization from a single location, as shown in the following table.

Task	Solr	OpenSearch
Administrative work	Solr Admin UI	OpenSearch Dashboards
Analytics	Grafana, Zeppelin, and other tools	OpenSearch Dashboards

The Solr Admin UI supports basic and Kerberos-based authentication. For authentication in OpenSearch, you need to create a valid role for accessing OpenSearch Dashboards and map users to the role. You can create different roles for different access requirements. For example, an administrative user might have access to all pages, but a read-only data user might see only specific indexes in OpenSearch Dashboards. For more information, see Defining users and roles in the OpenSearch documentation.

OpenSearch provides an ML-powered Assistant Toolkit for generating visualizations through natural language queries. In Amazon OpenSearch Service, you can use Amazon Q Developer for additional AI capabilities, including visualization generation, alert insights, query summaries,

anomaly detection recommendations, and chat support. For more information, see OpenSearch Assistant for OpenSearch Dashboards in the OpenSearch documentation and Amazon Q Developer for Amazon OpenSearch Service in the AWS documentation.

Monitoring

Effective Solr cluster monitoring combines native tools (Admin UI and health APIs), Java Management Extensions (JMX) metrics for real-time performance data, external systems such as Prometheus or Grafana for visualization, custom scripts for resource tracking, and automated alerts for critical issues. This multi-layered approach provides comprehensive visibility into cluster health, performance, and potential problems. When you use Amazon OpenSearch Service, you get access to the same functionality through the managed service.

Setting up effective monitoring is crucial for maintaining reliable and high-performing Amazon OpenSearch Service deployments. AWS provides several integrated tools that work together to create a comprehensive monitoring solution. This module outlines the key components for implementing monitoring for OpenSearch.

CloudWatch metrics

Amazon CloudWatch serves as the primary monitoring tool for Amazon OpenSearch Service. It provides real-time visibility into your OpenSearch resources, collects metrics at 60-second intervals (with some exceptions for EBS volumes at 5-minute intervals), and retains this data for two weeks. You can use CloudWatch to create custom dashboards and set up alerts, which makes it essential for proactive monitoring.

> **Note**
>
> Basic metrics are included at no additional cost with CloudWatch, but custom dashboards and alarms incur standard CloudWatch charges. For more information, see Amazon CloudWatch pricing.

CloudWatch Logs

CloudWatch Logs integration enables detailed logging capabilities for Amazon OpenSearch Service. Log types include:

- **Error logs** for troubleshooting OpenSearch service logs.
- **Search request slow logs** for tracking the total time a search request takes. Any search that takes longer than a set time limit is logged. You can turn this feature on and adjust the time

thresholds by using cluster settings. For more information, see Setting search request slow log thresholds in the Amazon OpenSearch Service documentation.

- **Shard slow logs** for performance monitoring. OpenSearch tracks slow operations by using two log types: search slow logs to monitor slow searches, and indexing slow logs to monitor slow indexing. It provides customizable time limits for each index to determine when an operation is considered slow. For more information, see Setting shard slow log thresholds in the Amazon OpenSearch Service documentation.
- **Audit logs** for tracking user actions on your clusters, such as logins, searches, and index changes. These logs can be customized to your needs. For more information, see Monitoring audit logs in the Amazon OpenSearch Service documentation.

Advanced monitoring

For additional monitoring features, use:

- Amazon EventBridge to set up automatic responses. For more information, see Monitoring OpenSearch Service events with Amazon EventBridge in the AWS documentation.
- AWS CloudTrail to track API activity and security. For more information, see Monitoring Amazon OpenSearch Service API calls with AWS CloudTrail in the AWS documentation.

These services work together to create a customized monitoring system for OpenSearch.

For general monitoring guidelines, see Operational best practices for Amazon OpenSearch Service in the AWS documentation. By following these guidelines and regularly reviewing and adjusting your monitoring setup, you can maintain a robust and effective monitoring system for your OpenSearch Service deployment. Remember to adapt these recommendations to your specific use case and requirements while maintaining alignment with AWS best practices.

Contributors

The following AWS experts contributed to this guide:

- Bharav Patel, Specialist Solutions Architect, OpenSearch, AWS
- Aswath Srinivasan, Senior Search Engine Architect, AWS
- John Trollinger, Principal Solutions Architect, AWS
- Vamsi Krishna Ganti, Senior Delivery Consultant, AWS
- Varun Sharma, Senior Delivery Consultant, AWS
- Prasad Mediboina, Senior Technical Account Manager, AWS

Resources

- Amazon OpenSearch Service
- Amazon OpenSearch Service Pricing
- Amazon OpenSearch Service Developer Guide
- Amazon OpenSearch Service API Reference
- OpenSearch API reference

For AWS sales and technical support, choose one of the links on the Contact AWS page.

Appendix: Solr sample schema

This appendix provides a sample schema in Solr for a product catalog. For a closer look at schema elements, see the Migrating your schema section earlier in this guide and Schema Elements in the Solr documentation.

```
<!-- Example Product Catalog Schema -->
<schema>
    <!-- Field Types -->
    <fieldType name="text_general" class="solr.TextField">
        <analyzer type="index">
            <tokenizer class="solr.StandardTokenizerFactory"/>
            <filter class="solr.LowerCaseFilterFactory"/>
            <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
            <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="15"/>
        </analyzer>
        <analyzer type="query">
            <tokenizer class="solr.StandardTokenizerFactory"/>
            <filter class="solr.StopFilterFactory" words="stopwords.txt"/>
            <filter class="solr.LowerCaseFilterFactory"/>
        </analyzer>
    </fieldType>

    <!-- Custom Field Types -->
    <fieldType name="custom_text_general" class="solr.TextField">
        <analyzer type="index">
            <tokenizer class="com.mycompany.CustomTokenizerFactory"/>
        </analyzer>
    </fieldType>

    <!-- Additional field type for sorting -->
    <fieldType name="string" class="solr.StrField"/>
    <fieldType name="float" class="solr.FloatPointField"/>

    <!-- Fields -->
    <field name="product_id" type="string" indexed="true" stored="true"/>
    <field name="price" type="float" indexed="true" stored="true"/>
    <field name="category" type="string" indexed="true" stored="true"/>
    <field name="brand" type="string" indexed="true" stored="true"/>
    <field name="title" type="text_general" indexed="true" stored="true"/>
```

```
    <field name="custom_title" type="custom_text_general" indexed="true" stored="true"/
>
    <field name="description" type="text_general" indexed="true" stored="true"/>

    <!-- Destination fields for copy operations -->
    <field name="text" type="text_general" indexed="true" stored="false"
 multiValued="true"/>

    <!-- Dynamic Fields -->
    <dynamicField name="attr_*" type="text_general" indexed="true" stored="true"/>

    <!-- Copy Fields Examples -->

    <!-- Unified search field - copy multiple fields to one destination -->
    <copyField source="title" dest="text"/>
    <copyField source="description" dest="text"/>
    <copyField source="brand" dest="text"/>
    <copyField source="category" dest="text"/>

    <!-- Unique Key -->
    <uniqueKey>product_id</uniqueKey>

</schema>
```

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	January 2, 2026

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

A2A (Agent-to-Agent)

A stateful protocol for agent-to-agent collaboration supporting task delegation and state transfer.

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

Agent

An AI system that can autonomously reason, plan, and take actions using tools to achieve goals.

Agent Ops

Operational practices for building, testing, deploying, and running AI agents in production at scale.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities.

For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

Citizen Developer

A business user who creates AI applications using no-code/low-code platforms without specialized technical skills.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in

an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.

- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

FM gateway

A centralized intermediary that controls and normalizes access to foundation models. Also known as an *LLM gateway*.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision

software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub CSPM, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

guardrails (AI)

Safety mechanisms that filter, validate, and constrain agent inputs and outputs to help ensure responsible and safe AI behavior.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver

high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

human-in-the-loop (HitL)

A workflow pattern where agent execution pauses for human review and approval at critical decision points.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage

Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

MCP

See Model Context Protocol.

Model Context Protocol (MCP)

A stateless protocol for agent-to-tool communication.

MCP server

A service that exposes one or more tools through the Model Context Protocol.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include

microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and

milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic `PUT` and `DELETE` requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends

setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

Shadow AI

Unauthorized AI applications built or used outside of governed channels within an organization.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

tool

A function or API that an agent can invoke to perform operations in external systems.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.