Database decomposition on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Database decomposition on AWS

Table of Contents

Database decomposition on AWS

Philippe Wanner and Saurabh Sharma, Amazon Web Services

October 2025 (document history)

Database modernization, particularly the decomposition of monolithic databases, is a critical workstream for organizations that want to improve agility, scalability, and performance in their data management systems. As businesses grow and their data needs become more complex, traditional monolithic databases often struggle to keep pace. This leads to performance bottlenecks, maintenance challenges, and difficulty adapting to changing business requirements.

The following are common challenges with monolithic databases:

- **Business domain misalignment** – Monolithic databases often fail to align technology with distinct business domains, which can limit organizational growth.
- **Scalability constraints** – Systems frequently hit scaling limits, which creates barriers to business expansion.
- **Architectural rigidity** – Tightly coupled structures make it difficult to update specific components without affecting the entire system.
- **Performance degradation** – Growing data loads and increasing user concurrency often lead to deteriorating system performance.

The following are the benefits of database decomposition:

- **Enhanced business agility** – Decomposition enables rapid adaptation to changing business needs and supports independent scaling.
- **Optimized performance** – Decomposition helps you create specialized database solutions that are tailored to specific use cases and independently scale each database.
- **Improved cost management** – Decomposition enables more efficient resource utilization and reduces operational costs.
- **Flexible licensing options** – Decomposition creates opportunities to transition from costly proprietary licenses to open source alternatives.
- **Innovation enablement** – Decomposition facilitates the adoption of purpose-built databases for specific workloads.

Intended audience

This guide helps database architects, cloud solutions architects, application development teams, and enterprise architects. It is designed to help you decompose monolithic databases into microservices-aligned data stores, implement domain-driven database architectures, plan database migration strategies, and scale database operations to meet growing business demands. To understand the concepts and recommendations in this guide, you should be familiar with relational and NoSQL database principles, AWS managed database services, and microservices architecture patterns. This guide is intended to help organizations that are in the initial stages of a database decomposition project.

Objectives

This guide can help your organization achieve the following objectives:

- Collect requirements for decomposing your target architecture.
- Develop a systematic methodology for evaluating risk and communicating.
- Create a decomposition plan.
- Define success metrics, key performance indicators (KPIs), a mitigation strategy, and a business continuity plan.
- Establish a better workload elasticity that helps you follow business demand.
- Learn how to adopt specialized databases for specific use cases, which enables innovation.
- Strengthen your organization's data security and governance.
- Reduce costs through the following:
  - Reduced licensing fees
  - Reduced vendor lock-in
  - Improved access to broader community support and innovations
  - Ability to choose different database technologies for different components
  - Gradual migration, which reduces risk and spreads costs over time
  - Improved resource utilization

Common challenges and managing responsibilities for database decomposition

Database decomposition is a complex process that requires careful planning, execution, and management. As organizations seek to modernize their data infrastructure, they often encounter a myriad of challenges that can impact the success of their projects. This section describes the common hurdles and introduces a structured approach to overcoming these obstacles.

Common challenges

A database decomposition project faces several challenges across technical, people, and business dimensions. On the technical front, ensuring data consistency across distributed systems poses a significant hurdle. It can also have potential performance and stability impacts during the transition period, and you must seamlessly integrate with existing systems. People-related challenges include the learning curve associated with the new system, potential resistance to change from employees, and the availability of necessary resources. From a business perspective, the project must contend with the risks of timeline overruns, budget constraints, and the potential for business disruption during the migration process.

Defining roles and responsibilities

Given these complex challenges that span technical, people, and business dimensions, establishing clear roles and responsibilities becomes critical for project success. A Responsible, Accountable, Consulted, and Informed (RACI) matrix provides the necessary structure to navigate these challenges. It explicitly defines who makes decisions, who performs the work, who provides input, and who needs to stay informed at each stage of the decomposition. This clarity helps prevent delays caused by ambiguous decision-making, encourages appropriate stakeholder engagement, and creates accountability for key deliverables. Without such a framework, teams may struggle with overlapping responsibilities, missed communications, and unclear escalation paths—issues that could exacerbate the existing technical complexities and change management challenges while increasing the risk of timeline and budget overruns.

The following sample RACI matrix is a starting point that can help you clarify potential roles and responsibilities in your organization.

Task or activity	Project manager	Architect	Developer	Stakeholder
Identify business outcomes and challenges	A/R	R	C	–
Define the scope and identify requirements	A	R	C	C/I
Identify the project success metrics	A	R	C	I
Create and execute the communication plan	A/R	C	C	I
Define the target architecture	I	A/R	C	–
Control database access	I	A/R	R	–
Create and execute the business continuity plan	A/R	C	I	–
Analyze cohesion and coupling	I	A/R	R	I
Move the business logic (such as stored	I	A	R	–

procedure s) from the database to the application layer				
Decouple table relationships, known as *joins*	I	A	R	–

Defining the scope and requirements for database decomposition

When you define the scope and identify requirements for your database decomposition project, you must work backward from your organization's needs. This requires a systematic approach that balances technical feasibility with business value. This initial step sets the foundation for the entire process and helps you make sure that the project's objectives align with the organization's goals and capabilities.

This section contains the following topics:

Establishing a core analysis framework

The scope definition begins with a systematic workflow that guides the analysis through four interconnected phases. This comprehensive approach makes sure that database decomposition efforts are grounded in a thorough understanding of the existing systems and operational requirements. The following are the phases in the core analysis framework:

1. **Actor analysis** – Thoroughly identify all systems and applications that interact with the database. This involves mapping both producers that perform write operations and consumers that handle read operations, while documenting their access patterns, frequencies, and peak usage times. This customer-centric view helps you understand the impact of any changes and identify critical paths that require special attention during decomposition.
2. **Activity analysis** – Dive deep into the specific operations that each actor performs. You create detailed create, read, update, and delete (CRUD) matrices for each system and identify which tables they access and how. This analysis helps you discover natural boundaries for decomposition and highlights areas where you can simplify the current architecture.

3. **Dependency mapping** – Document both direct and indirect dependencies between systems, creating clear visualizations of data flows and relationships. This helps identify potential breaking points and areas where careful planning is needed to earn trust. The analysis considers both technical dependencies, such as shared tables and foreign keys, and business process dependencies, such as workflow sequences and reporting requirements.
4. **Consistency requirements** – Examine each operation's consistency needs with high standards. Determine which operations require immediate consistency, such as financial transactions. Other operations can operate with eventual consistency, such as analytics updates. This analysis directly influences the choice of decomposition patterns and architectural decisions throughout the project.

Defining system boundaries for database decomposition

System boundaries are logical perimeters that define where one system ends and another begins, encompassing data ownership, access patterns, and integration points. When defining system boundaries, make thoughtful but decisive choices that balance comprehensive planning with practical implementation needs. Consider the database as a logical unit that might span multiple physical databases or schemas. This boundary definition accomplishes the following critical objectives:

- Identifies all external actors and their interaction patterns
- Comprehensively maps both inbound and outbound dependencies
- Documents technical and operational constraints
- Clearly delineates the scope of the decomposition effort

Considering release cycles

Understanding release cycles is crucial for planning database decomposition. Review the renewal times for both the target system and any dependent systems. Identify opportunities for coordinated changes. Consider any planned decommissioning of connected systems because this might influence your decomposition strategy. Factor in existing change windows and deployment constraints to minimize business disruption. Make sure that your implementation plan aligns with release schedules across all connected systems.

Evaluating technical constraints for database decomposition

Before proceeding with database decomposition, assess the key technical limitations that will shape your modernization approach. Examine the capabilities of your current technology stack, including database versions, frameworks, performance requirements, and service level agreements. Consider security and compliance mandates, especially for regulated industries. Review current data volumes, growth projections, and available migration tools to inform your scaling decisions. Finally, confirm your access rights to source code and system modifications because these will determine the viable decomposition strategies.

Understanding organizational context

Successful database decomposition requires that you understand the broader organizational landscape in which the system operates. Map cross-departmental dependencies, and establish clear communication channels between teams. Assess your team's technical capabilities, and identify any training needs or skill gaps that you need to address. Consider change management implications, including how to manage transitions and maintain business continuity. Evaluate available resources and any constraints, such as budget or staffing limitations. Finally, align your decomposition strategy with stakeholder expectations and priorities to promote continued support throughout the project.

Assessing risk for database decomposition

A comprehensive risk assessment is essential for database decomposition success. Carefully evaluate risks, such as data integrity during the migration, potential system performance degradation, possible integration failures, and security vulnerabilities. These technical challenges must be balanced against business risks, including potential operational disruptions, resource limitations, timeline delays, and budget constraints. For each identified risk, develop specific mitigation strategies and contingency plans in order to maintain project momentum while protecting business operations.

Create a risk matrix that evaluates both impact and probability of potential issues. Work with technical teams and business stakeholders to identify risks, set clear thresholds for intervention, and develop specific mitigation strategies. For example, rate data loss risk as high impact and low probability, and it requires robust backup strategies. Minor performance degradation might be medium impact and high probability, and it requires proactive monitoring.

Establish regular risk review cycles to reassess priorities and adjust mitigation plans as the project evolves. This systematic approach makes sure that resources are focused on the most critical risks while maintaining clear escalation paths for emerging issues.

Defining success criteria for database decomposition

Success criteria for database decomposition must be clearly defined and measurable across multiple dimensions. From a business perspective, establish specific targets for cost reduction, improved time-to-market, system availability, and customer satisfaction. Technical success should be measured through quantifiable improvements in system performance, deployment efficiency, data consistency, and overall reliability. For the migration process, define strict requirements for zero data loss, acceptable business disruption limits, budget compliance, and timeline adherence.

Document these criteria thoroughly by maintaining baseline and target metrics, clear measurement methodologies, and regular review schedules. Assign clear owners for each success metric, and map dependencies between different metrics. This comprehensive approach to measuring success aligns technical achievements with business outcomes, while maintaining accountability throughout the decomposition journey.

Controlling database access during decomposition

Many organizations face a common scenario: a central database that has grown organically over many years and is accessed directly by multiple services and teams. This creates several critical problems:

- **Uncontrolled growth** – As teams continuously add new features and modify schemas, the database becomes increasingly complex and difficult to manage.
- **Performance concerns** – Even with hardware improvements, the growing load eventually threatens to exceed the database's capabilities. Impossibility to tune queries due to schema complexity or lack of skills. Unable to predict or explain system performance.
- **Decomposition paralysis** – It becomes nearly impossible to split or refactor the database while it's actively being modified by multiple teams.

> **Note**
>
> Monolithic database systems often reuse the same credentials for applications or services or for administration. This leads to poor database traceability. Setting dedicated roles and adopting the principle of least privilege can help you increase security and availability.

When dealing with a monolithic database that has become unwieldy, one of the most effective patterns to control access is called a *database wrapper service*. It provides a strategic first step in managing complex database systems. It establishes controlled database access and enables gradual modernization, while reducing risk. This approach creates a foundation for incremental improvements by providing clear visibility into data usage patterns and dependencies. It's a transitional architecture that serves as a step toward full database decomposition. The wrapper service provides the stability and control needed to make that journey successfully.

This section contains the following topics:

Controlling access with the database wrapper service pattern

A *wrapper service* is a service layer that acts as a facade for the database. This approach is particularly valuable when you need to maintain existing functionality while preparing for future decomposition. This pattern follows a simple principle—when something is too messy, start by containing the mess. The wrapper service becomes the only authorized way to access the database, providing a controlled interface while hiding the underlying complexity.

Use this pattern when immediate database decomposition isn't feasible due to complex schemas or when multiple services require continuous data access. It's particularly valuable during transition periods because it provides time for careful refactoring while maintaining system stability. The pattern works well when consolidating data ownership to specific teams or when new applications need aggregated views across multiple tables.

For example, apply this pattern when:

- Schema complexity prevents immediate separation
- Multiple teams need ongoing data access
- Gradual modernization is preferred
- Team restructuring requires clear data ownership
- New applications need consolidated data views

Benefits and limitations of the database wrapper service pattern

The following are the benefits of the database wrapper pattern:

- **Controlled growth** – The wrapper service prevents further uncontrolled additions to the database schema.
- **Clear boundaries** – The implementation process helps you establish clear ownership and responsibility boundaries.
- **Refactoring freedom** – A wrapper service lets you make internal changes without impacting consumers.
- **Improved observability** – A wrapper service is a single point for monitoring and logging.
- **Simplified testing** – A wrapper service makes it easier for consuming services to create simplified, mock versions for testing.

The following are the limitations of the database wrapper pattern.

- **Technology coupling** – A wrapper service works best when it uses the same technology stack as the consuming services.
- **Initial overhead** – The wrapper service requires additional infrastructure that might affect performance.
- **Migration effort** – To implement the wrapper service, you must coordinate across teams to transition away from direct access.
- **Performance** – If the wrapping service experiences high traffic, heavy usage, or frequent access, consuming services might experience poor performance. On top of the database, the wrapper service must handle pagination, cursors, and database connections. Depending it on your use case, it might not scale well, and it might be a poor fit for extract, transform, and load (ETL) workloads.

Implementing the database wrapper service pattern

There are two phases to implement the database wrapper service pattern. First, you create the database wrapper service. Then, you direct all access through it and document the access patterns.

Phase 1: Creating the database wrapper service

Create a lightweight service layer that acts as a gatekeeper to your database. Initially, it should mirror all existing functionalities. This wrapper service becomes the mandatory access point for all database operations, which converts direct database dependencies into service-level dependencies. Implement detailed logging and monitoring at this layer to track usage patterns, performance metrics, and access frequencies. Maintain your existing stored procedures, but make sure that they're accessed only through this new service interface.

Phase 2: Implementing access control

Systematically redirect all database access through the wrapper service, and then revoke direct database permissions from external systems that access the database directly. Document each access pattern and dependency as services are migrated. This controlled access enables internal refactoring of database components without disrupting external consumers. For example, start with low-risk, read-only operations instead of complex transactional workflows.

Phase 3: Monitor database performance

Use the wrapper service as a centralized monitoring point for database performance. Track key metrics, including query response times, usage patterns, error rates, and resource utilization. Set up alerts for performance thresholds and unusual patterns. For example, monitor slow-running queries, connection pool utilization, and transaction throughput to proactively identify potential issues.

Use this consolidated view to optimize database performance through query tuning, resource allocation adjustments, and usage pattern analysis. The centralized nature of the wrapper service makes it easier to implement improvements and validate their impact across all consumers, while maintaining consistent performance standards.

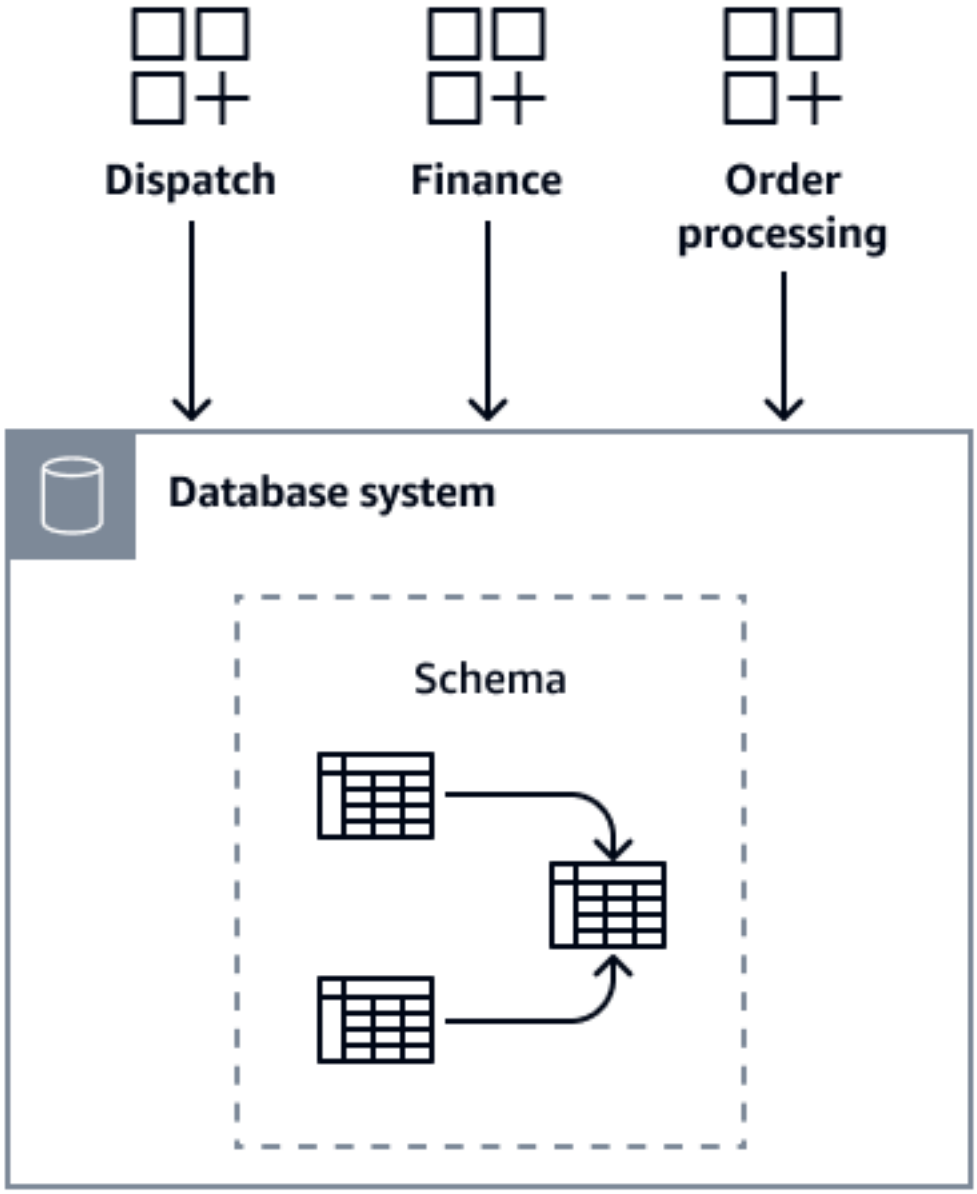
Best practices for implementing a database wrapper service

The following best practices can help you implement a database wrapper service:

- **Start small** – Begin with a minimal wrapper that simply proxies existing functionality
- **Maintain stability** – Keep the service interface stable while making internal improvements
- **Monitor usage** – Implement comprehensive monitoring to understand access patterns
- **Clear ownership** – Assign a dedicated team to maintain both the wrapper and the underlying schema
- **Encourage local storage** – Motivate teams to store their data in their own databases

Scenario-based example

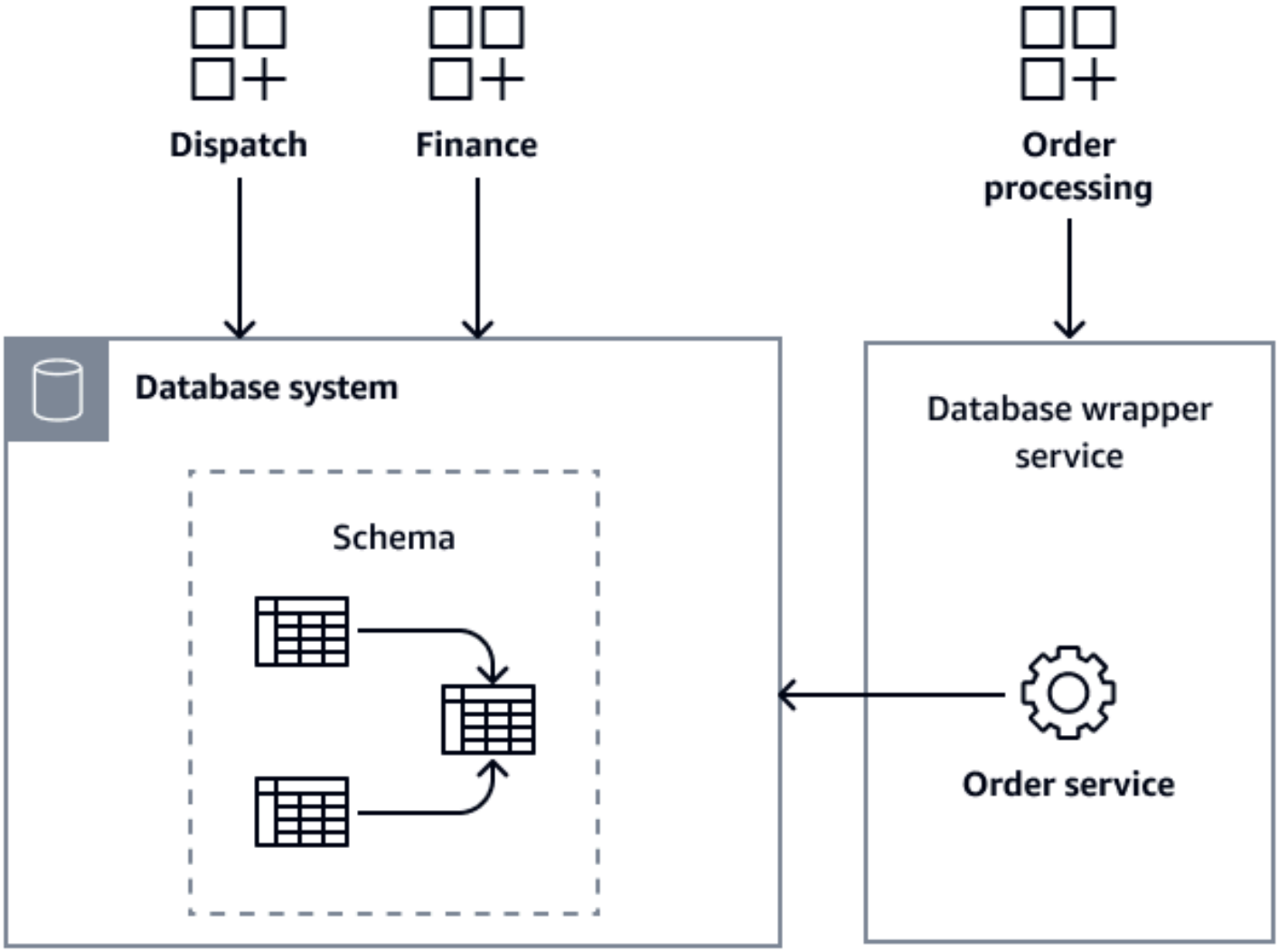
This section describes an example of how a fictitious company, named *AnyCompany Books*, could use the database wrapper pattern to control access to their monolithic database system. At AnyCompany Books, there are three critical services: Dispatch, Finance, and Order Processing. These services share access to a central database. Each service is maintained by a different team. Over time, they independently modify the database schema to meet their specific needs. This has led to a tangled web of dependencies and an increasingly complex database structure.



The company's application or enterprise architect recognizes the need to decompose this monolithic database. Their goal is to give each service its own dedicated database to improve maintainability and reduce cross-team dependencies. However, they face a significant challenge—it's nearly impossible to decompose the database while all three teams continue to actively modify it for their ongoing projects. The constant schema changes and lack of coordination between teams make it extremely risky to attempt any significant restructuring.

The architect uses the database wrapper service pattern to start controlling access to the monolithic database. First, they set up the database wrapper service for a particular module, called

the Order service. Then, they redirect the Order Processing service to access the wrapper service instead of directly accessing the database. The following image shows the modified infrastructure.

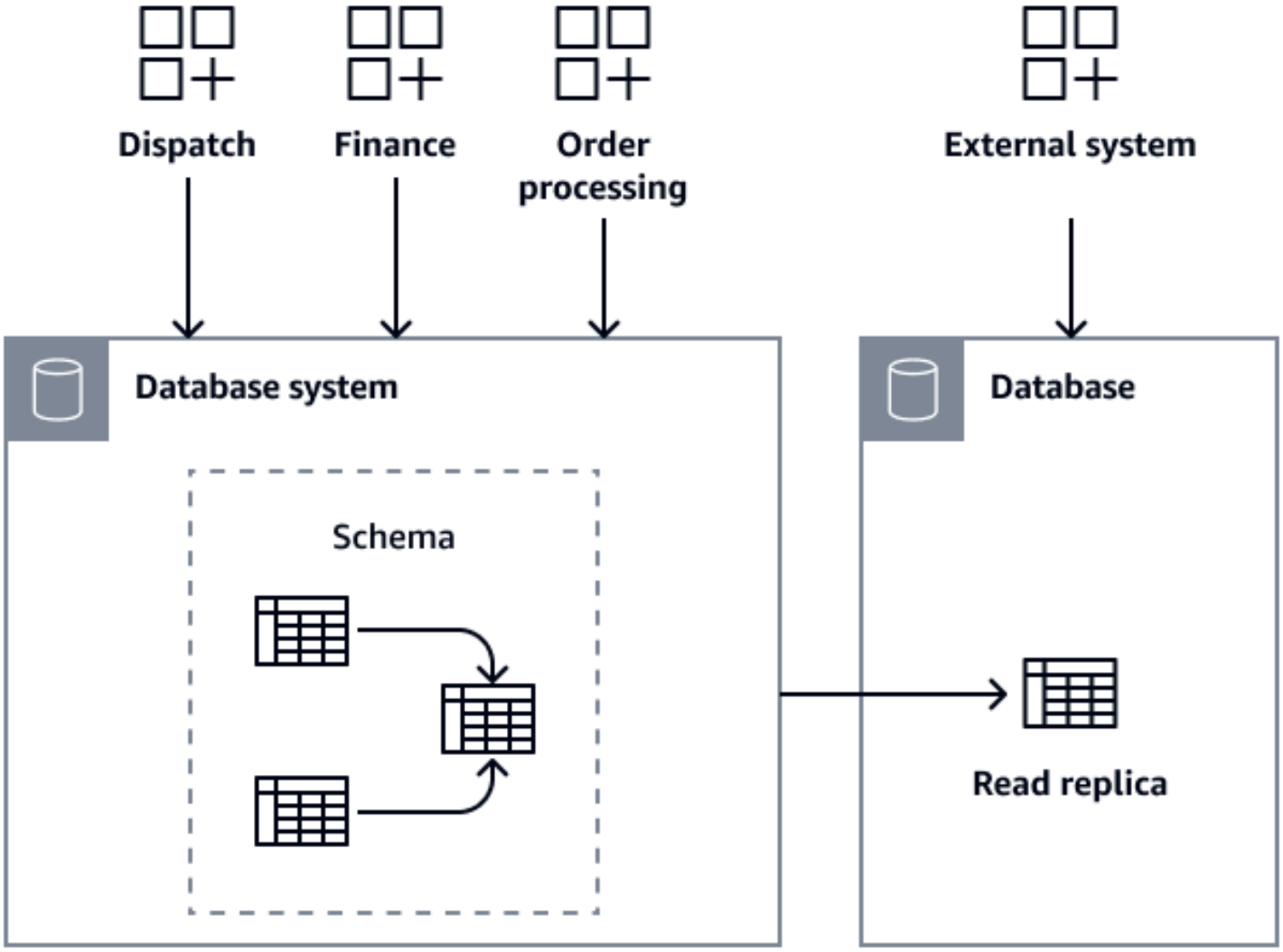


Controlling access with the CQRS pattern

Another pattern that you can use to isolate external systems that connect to this central database is *command query responsibility segregation (CQRS)*. If some of the external systems are connecting to your central database primarily for reads, such as analytics, reporting, or other read-intensive operations, you can create separate read-optimized data stores.

This pattern effectively isolates these external systems from the impacts of database decomposition and schema changes. By maintaining dedicated read replicas or purpose-built data stores for specific query patterns, teams can continue their operations without being affected by changes in the primary database structure. For example, while you decompose your monolithic

database, reporting systems can continue to work with their existing data views, and analytical workloads can maintain their current query patterns through dedicated analytical stores. This approach provides technical isolation and enables organizational autonomy because different teams can evolve their systems independently without tight coupling to the primary database's transformation journey.



For more information about this pattern and an example of its use to decouple table relationships, see CQRS pattern later in this guide.

Analyzing cohesion and coupling for database decomposition

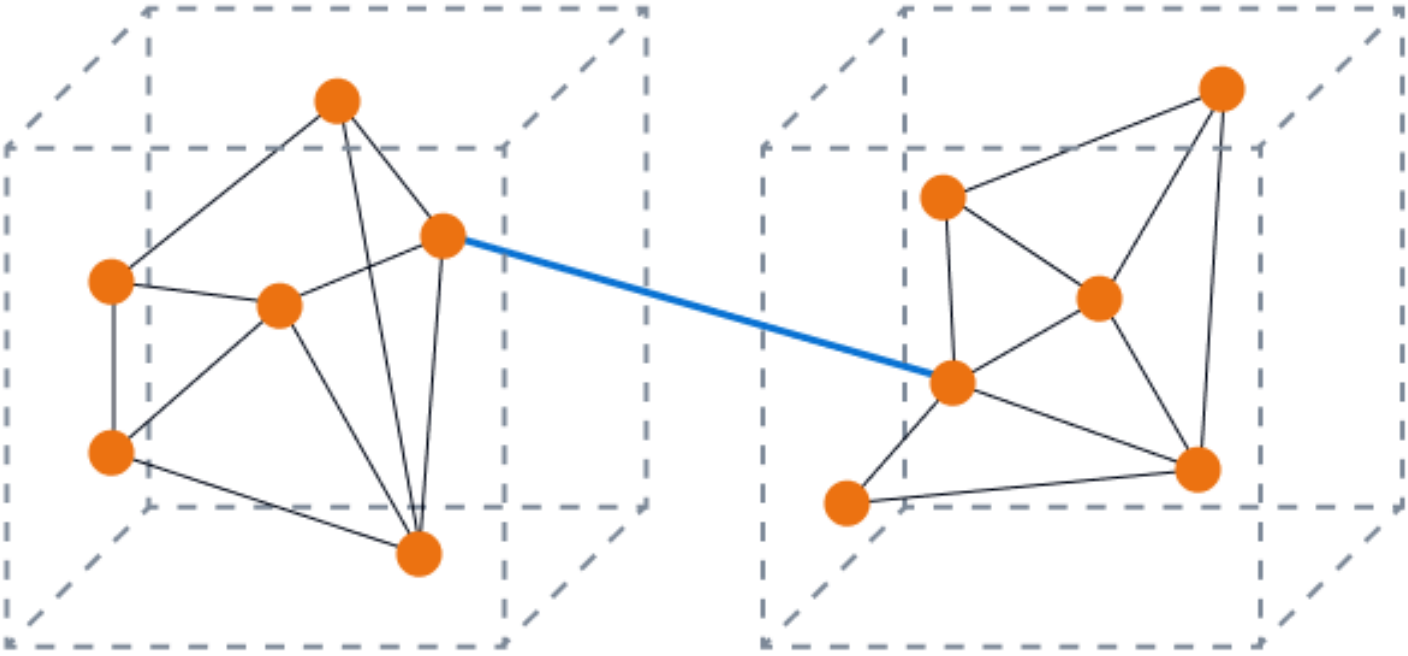
This section helps you analyze coupling and cohesion patterns in your monolithic database to guide its decomposition. Understanding how database components interact and depend on each other is crucial for identifying natural break points, assessing complexity, and planning a phased migration approach. This analysis reveals hidden dependencies, highlights areas that are suitable for immediate separation, and helps you prioritize decomposition efforts while minimizing transformation risks. By examining both coupling and cohesion, you can make informed decisions about the component separation sequence in order to maintain system stability throughout the transformation process.

This section contains the following topics:

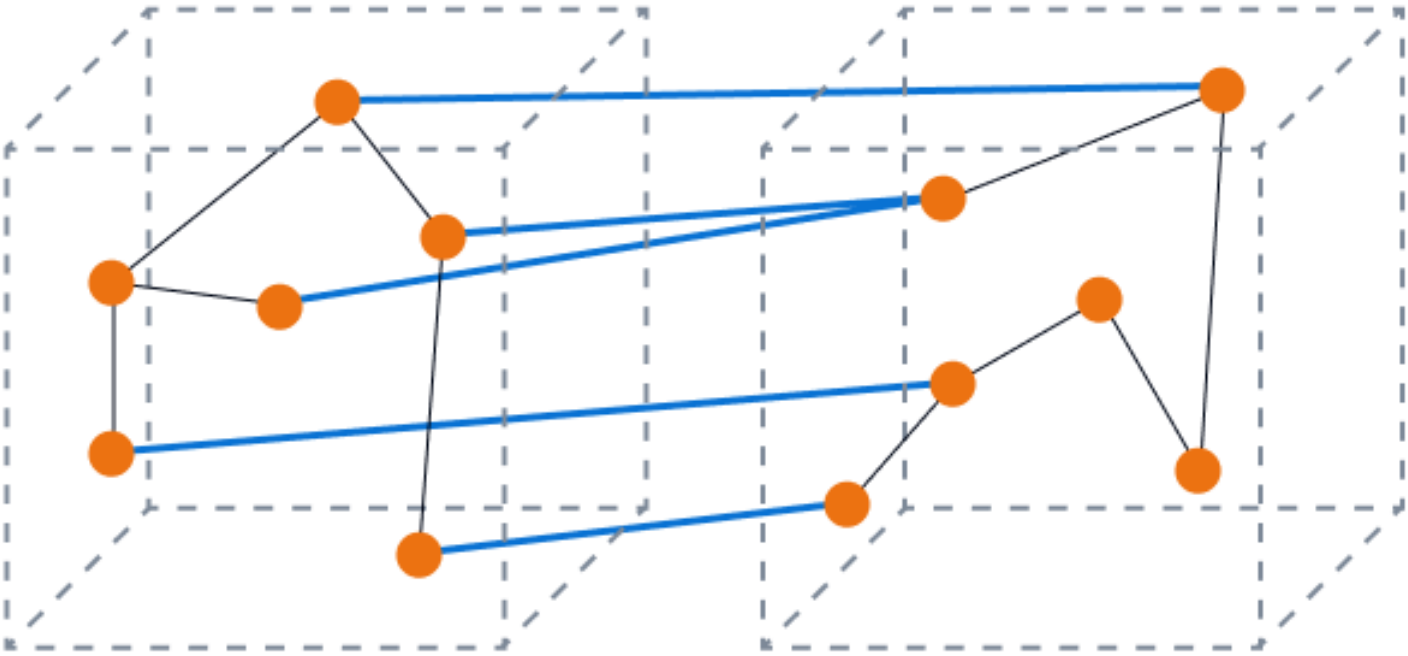
About cohesion and coupling

Coupling measures the degree of interdependence between database components. In a well-designed system, you want to achieve loose coupling, where changes to one component have minimal impact on others. *Cohesion* measures how well the elements within a database component work together to serve a single, well-defined purpose. High cohesion indicates that a component's elements are strongly related and focused on a specific function. When decomposing a monolithic database, you must analyze both the cohesion within individual components and the coupling between them. This analysis helps you make informed decisions about how to break down the database while maintaining system integrity and performance.

The following image shows loose coupling with high cohesion. The components in the database work together to perform a specific function, and you minimize the impact of change on a single component. This is the ideal state.



The following image shows high coupling with low cohesion. The database components are disconnected, and changes are highly likely to impact other components.



Common coupling patterns in monolithic databases

There are several coupling patterns that are commonly found when decomposing a monolithic database into microservice-specific databases. Understanding these patterns is crucial for

successful database modernization initiatives. This section describes each pattern, its challenges, and best practices for reducing coupling.

Implementation coupling pattern

Definition: Components are tightly interconnected at the code and schema level. For example, modifying the structure of a `customer` table impacts `order`, `inventory`, and `billing` services.

Modernization impact: Each microservice requires its own dedicated database schema and data access layer.

Challenges:

- Changes to shared tables affect multiple services
- High risk of unintended side effects
- Increased testing complexity
- Difficult to modify individual components

Best practices to reduce coupling:

- Define clear interfaces between components
- Use abstraction layers to hide implementation details
- Implement domain-specific schemas

Temporal coupling pattern

Definition: Operations must run in a specific sequence. For example, order processing cannot proceed until inventory updates are complete.

Modernization impact: Each microservice needs autonomous data control.

Challenges:

- Breaking synchronous dependencies between services
- Performance bottlenecks
- Difficult to optimize

- Limited parallel processing

Best practices to reduce coupling:

- Implement asynchronous processing where possible
- Use event-driven architectures
- Design for eventual consistency when appropriate

Deployment coupling pattern

Definition: System components must be deployed as a single unit. For example, a minor change to the payment processing logic requires redeploying the entire database.

Modernization impact: Independent database deployments per service

Challenges:

- High-risk deployments
- Limited deployment frequency
- Complex rollback procedures

Best practices to reduce coupling:

- Break down into independently deployable components
- Implement database sharding strategies
- Use blue-green deployment patterns

Domain coupling pattern

Definition: Business domains share database structures and logic. For example, the `customer`, `order`, and `inventory` domains share tables and stored procedures.

Modernization impact: Domain-specific data isolation

Challenges:

- Complex domain boundaries

- Difficult to scale individual domains
- Tangled business rules

Best practices to reduce coupling:

- Identify clear domain boundaries
- Separate data by domain context
- Implement domain-specific services

Common cohesion patterns in monolithic databases

There are several cohesion patterns that are commonly found when evaluating database components for decomposition. Understanding these patterns is crucial for identifying well-structured database components. This section describes each pattern, its characteristics, and best practices for strengthening cohesion.

Functional cohesion pattern

Definition: All elements directly support and contribute to performing a single, well-defined function. For example, all stored procedures and tables in a payment-processing module handle only payment-related operations.

Modernization impact: Ideal pattern for microservice database design

Challenges:

- Identifying clear functional boundaries
- Separating mixed-purpose components
- Maintaining single responsibility

Best practices to strengthen cohesion:

- Group related functions together
- Remove unrelated functionality
- Define clear component boundaries

Sequential cohesion pattern

Definition: Output from one element becomes input for another. For example, validation results for an order feed into order processing.

Modernization impact: Requires careful workflow analysis and data flow mapping

Challenges:

- Managing dependencies between steps
- Handling failure scenarios
- Maintaining process order

Best practices to strengthen cohesion:

- Document clear data flows
- Implement proper error handling
- Design clear interfaces between steps

Communicational cohesion pattern

Definition: Elements operate on the same data. For example, customer-profile management functions all work with customer data.

Modernization impact: Helps identify data boundaries for service separation to decrease coupling between modules

Challenges:

- Determining data ownership
- Managing shared data access
- Maintaining data consistency

Best practices to strengthen cohesion:

- Define clear data ownership
- Implement proper data access patterns

- Design effective data partitioning

Procedural cohesion pattern

Definition: Elements are grouped together because they must be executed in a specific order, but they may not be functionally related. For example, in order processing, a stored procedure that handles both order validation and user notification is grouped together simply because they happen in sequence, even though they serve different purposes and could be handled by separate services.

Modernization impact: Requires careful separation of procedures while maintaining process flow

Challenges:

- Maintaining correct process flow after decomposition
- Identifying true functional boundaries compared to procedural dependencies

Best practices to strengthen cohesion:

- Separate procedures based on their functional purpose rather than execution order
- Use orchestration patterns to manage process flow
- Implement workflow management systems for complex sequences
- Design event-driven architectures to handle process steps independently

Temporal cohesion pattern

Definition: Elements are related by timing requirements. For example, when an order is placed, several operations must execute together: inventory check, payment processing, order confirmation, and shipping notification must all occur within a specific time window to maintain a consistent order state.

Modernization impact: Might require special handling in distributed systems

Challenges:

- Coordinating timing dependencies across distributed services
- Managing distributed transactions

- Confirming process completion across multiple components

Best practices to strengthen cohesion:

- Implement proper scheduling mechanisms and timeouts
- Use event-driven architectures with clear sequence handling
- Design for eventual consistency with compensation patterns
- Implement saga patterns for distributed transactions

Logical or coincidental cohesion pattern

Definition: Elements are logically categorized to do the same things, even though they have weak or no meaningful relationships. An example is storing customer order data, warehouse inventory counts, and marketing email templates in the same database schema because they all relate to sales operations, despite having different access patterns, lifecycle management, and scaling requirements. Another example is combining order payment processing and product catalog management within the same database component because they're both part of the e-commerce system, even though they serve distinct business functions with different operational needs.

Modernization impact: Should be refactored or reorganized

Challenges:

- Identifying better organization patterns
- Breaking unnecessary dependencies
- Restructuring components that were arbitrarily grouped

Best practices to strengthen cohesion:

- Reorganize based on true functional boundaries and business domains
- Remove arbitrary groupings based on superficial relationships
- Implement proper separation of elements based on business capabilities
- Align database components with their specific operational requirements

Implementing low coupling and high cohesion

Best practices

The following best practices can help you achieve low coupling:

- Minimize dependencies between database components
- Use well-defined interfaces for component interaction
- Avoid shared state and global data structures

The following best practices can help you achieve high cohesion:

- Group related data and operations together
- Make sure that each component has a single, clear responsibility
- Maintain clear boundaries between different business domains

Phase 1: Map data dependencies

Map data relationships and identify natural boundaries. You can use tools, such as SchemaSpy, to visualize the database by showing the tables in entity-relationship (ER) diagram. This provides a static analysis of the database and indicates some of the clear boundaries and dependencies within the database.

You can also export your database schemas in a graph database or in a Jupiter notebook. Then, you can apply clustering or interconnected components algorithms to identify natural boundaries and dependencies. Other AWS Partner tools, such as CAST Imaging, can help to understand your database dependencies.

Phase 2: Analyze transaction boundaries and access patterns

Analyze transaction patterns to maintain atomicity, consistency, isolation, durability (ACID) properties and understand how data is accessed and modified. You can use database analysis and diagnosis tools, such as Oracle Automatic Workload Repository (AWR) or PostgreSQL pg_stat_statements. This analysis helps you understand who is accessing the database and what the transaction boundaries are. It can also help you understand the cohesion and coupling between tables at runtime. You can also use monitoring and profiling tools that can link code and database execution profiles, such as Dynatrace AppEngine.

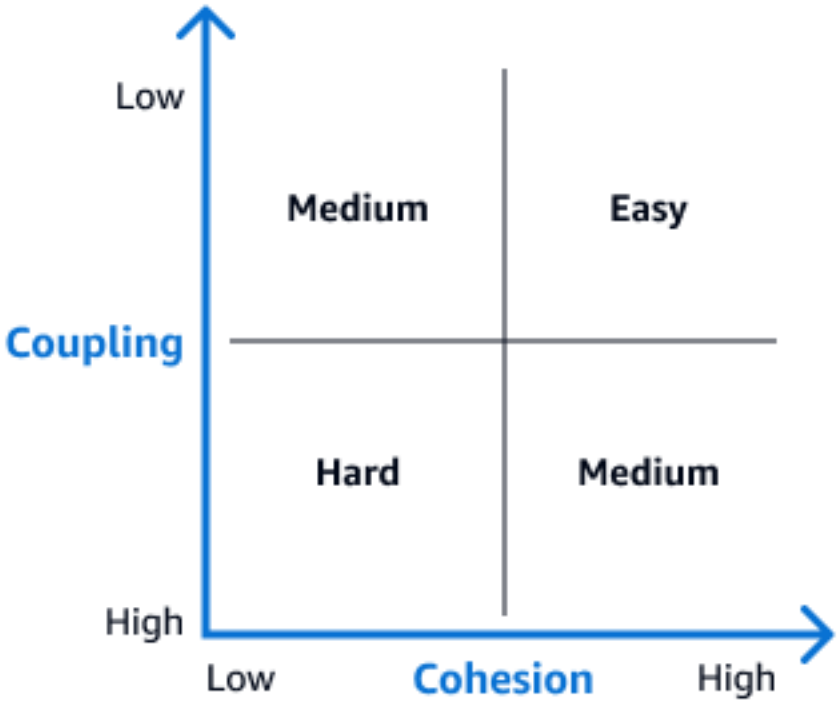
AI tools, such as vFunction, can help you identify domain boundaries by analyzing the application's functional and domain boundaries. Although vFunction primarily analyzes the application layer, its insights can guide the decomposition of both the application and the database, supporting alignment with business domains.

Phase 3: Identify self-contained tables

Look for tables that demonstrate two key characteristics:

- High cohesion – The table's contents are strongly related to each other
- Low coupling – They have minimal dependencies on other tables.

The following coupling-cohesion matrix can help you identify the difficulty of decoupling each table. Tables that appear in the upper-right quadrant of this matrix are ideal candidates for initial decoupling efforts because they're the easiest to separate. In an ER diagram, these tables have few foreign key relationships or other dependencies. After you have decoupled these tables, progress toward tables with more complex relationships.



Note

Database structure often mirrors application architecture. Tables that are easier to decouple at the database level typically correspond to components that are easier to convert into microservices at the application level.

Migrating business logic from the database to the application layer

Migrating business logic from database-stored procedures, triggers, and functions to application-layer services is a critical step in decomposing monolithic databases. This transformation improves service autonomy, simplifies maintenance, and enhances scalability. This section provides guidance on analyzing database logic, planning the migration strategy, and then implementing the transformation while maintaining business continuity. It also discusses establishing an effective rollback plan.

This section contains the following topics:

Phase 1: Analyzing the business logic

When modernizing monolithic databases, you must first conduct a comprehensive analysis of your existing database logic. This phase focuses on three primary categories:

- *Stored procedures* often contain critical business operations, including data-manipulation logic, business rules, validation checks, and calculations. As core components of the application business logic, they require careful decomposition. For instance, a financial organization's stored procedures might handle interest calculations, account reconciliation, and compliance checks.
- *Triggers* are key database components that handle audit trails, data validation, calculations, and cross-table consistency. For example, a retail organization might use triggers to manage inventory updates throughout their order processing system, which demonstrates the complexity of automated database operations.
- *Functions* in databases primarily manage data transformations, calculations, and lookup operations. They are often embedded across multiple procedures and applications. For example, a healthcare organization might use functions to normalize patient data or look up medical codes.

Each category represents different aspects of business logic that is embedded within the database layer. You need to carefully evaluate and plan each in order migrate them to the application layer.

During this analysis phase, customers typically face three significant challenges. First, complex dependencies emerge through nested procedure calls, cross-schema references, and implicit data dependencies. Second, transaction management becomes critical, particularly when dealing with multi-step transactions and maintaining data consistency across distributed systems. Third, performance considerations must be carefully evaluated, especially for batch processing operations, bulk data updates, and real-time calculations that currently benefit from being close to the data.

To effectively address these challenges, you can use AWS Schema Conversion Tool (AWS SCT) for initial analysis and then use detailed dependency-mapping tools. This approach helps you understand the full scope of your database logic and create a comprehensive migration strategy that maintains business continuity during decomposition.

By thoroughly understanding these components and challenges, you can better plan your modernization journey and make informed decisions about which elements to prioritize during the migration to a microservices-based architecture.

When analyzing database code components, create comprehensive documentation for each stored procedure, trigger, and function. Start by clearly describing its purpose and core functionality, including business rules it implements. Detail all input and output parameters, and note their data types and valid ranges. Map out dependencies on other database objects, external systems, and downstream processes. Clearly define transaction boundaries and isolation requirements to maintain data integrity. Document any performance expectations, including response time requirements and resource utilization patterns. Finally, analyze usage patterns to understand peak loads, frequency of execution, and critical business periods.

Phase 2: Classifying the business logic

Effective database decomposition requires systematic categorization of database logic across key dimensions: complexity, business impact, dependencies, usage patterns, and migration difficulty. This classification helps you identify high-risk components, determine testing requirements, and establish migration priorities. For example, complex stored procedures with high business impact and frequent usage require careful planning and extensive testing. However, simple, rarely used functions with minimal dependencies might be suitable for early migration phases.

This structured approach creates a balanced migration roadmap that minimizes business disruption while maintaining system stability. By understanding these interrelationships, you can improve the sequence of your decomposition efforts and allocate resources appropriately.

Phase 3: Migrating the business logic

After you have analyzed and classified your business logic, it is time to migrate it. There are two approaches when migrating business logic out of a monolithic database: move the database logic to the application layer, or move the business logic to another database that is part of the microservice.

If you migrate the business logic to the application, then the database tables store only the data, and the database doesn't contain any business logic. This is the recommended approach. You can use Ispirer or generative AI tools, such as Amazon Q Developer or Kiro, to convert database business logic for the application layer, such as conversion to Java. For more information, see Migrate business logic from database to application for faster innovation and flexibility (AWS blog post).

If you migrate the business logic to another database, you can use AWS Schema Conversion Tool (AWS SCT) to convert existing database schemas and code objects to your target database. It supports purpose-built AWS database services, such as Amazon DynamoDB, Amazon Aurora, and Amazon Redshift. By providing a comprehensive assessment report and automated conversion capabilities, AWS SCT helps streamline the transition process, allowing you to focus on optimizing your new database structure for improved performance and scalability. As you progress through your modernization project, AWS SCT can handle incremental conversions to support a phased approach, enabling you to validate and fine-tune each step of your database transformation.

Rollback strategy for business logic

Two critical aspects of any decomposition strategy are maintaining backward compatibility and implementing comprehensive rollback procedures. These elements work together to help protect operations during the transition period. This section describes how to manage compatibility during the decomposition process and establish effective emergency rollback capabilities that safeguard against potential issues.

Maintain backward compatibility

During database decomposition, maintaining backward compatibility is essential for smooth transitions. Keep existing database procedures temporarily in place while gradually implementing new functionality. Use version control to track all changes and manage multiple database versions simultaneously. Plan for an extended coexistence period where both the source and target systems must operate reliably. This provides time to test and validate the new system before retiring legacy components. This approach minimizes business disruption and provides a safety net for rollback if needed.

Emergency rollback plan

A comprehensive rollback strategy is essential for safe database decomposition. Implement feature flags in your code to control which version of business logic is active. This allows you to instantly switch between the new and original implementations without deployment changes. This approach provides fine-grained control over the transition and helps you roll back quickly if issues arise. Keep the original logic as a verified backup, and maintain detailed rollback procedures that specify triggers, responsibilities, and recovery steps.

Regularly test these rollback scenarios under various conditions to validate their effectiveness, and make sure that teams are familiar with emergency procedures. Feature flags also enable gradual rollouts by selectively enabling new functionality for specific user groups or transactions. This provides an additional layer of risk mitigation during the transition.

Decoupling table relationships during database decomposition

This section provides guidance on breaking down complex table relationships and JOIN operations during monolithic database decomposition. A table *join* combines rows from two or more tables based on a related column between them. The goal of separating these relationships is to reduce high coupling between tables while maintaining data integrity across microservices.

This section contains the following topics:

Denormalization strategy

Denormalization is a database design strategy that involves intentionally introducing redundancy by combining or duplicating data across tables. When breaking apart a large database into small databases, it might make sense to duplicate some data across services. For example, storing basic customer details, such as name and email addresses, in both a marketing service and an order service eliminates the need for constant cross-service lookups. The marketing service might need customer preferences and contact information for campaign targeting, while the order service requires the same data for order processing and notifications. While this creates some data redundancy, it can significantly improve service performance and independence, allowing the marketing team to operate their campaigns without depending on real-time customer service lookups.

When implementing denormalization, focus on frequently accessed fields that you identify through careful analysis of data access patterns. You can use tools, such Oracle AWR reports or pg_stat_statements, to understand which data is commonly retrieved together. Domain experts can also provide valuable insights into natural data groupings. Remember that denormalization

isn't an all-or-nothing approach—only duplicate data that demonstrably improves system performance or reduces complex dependencies.

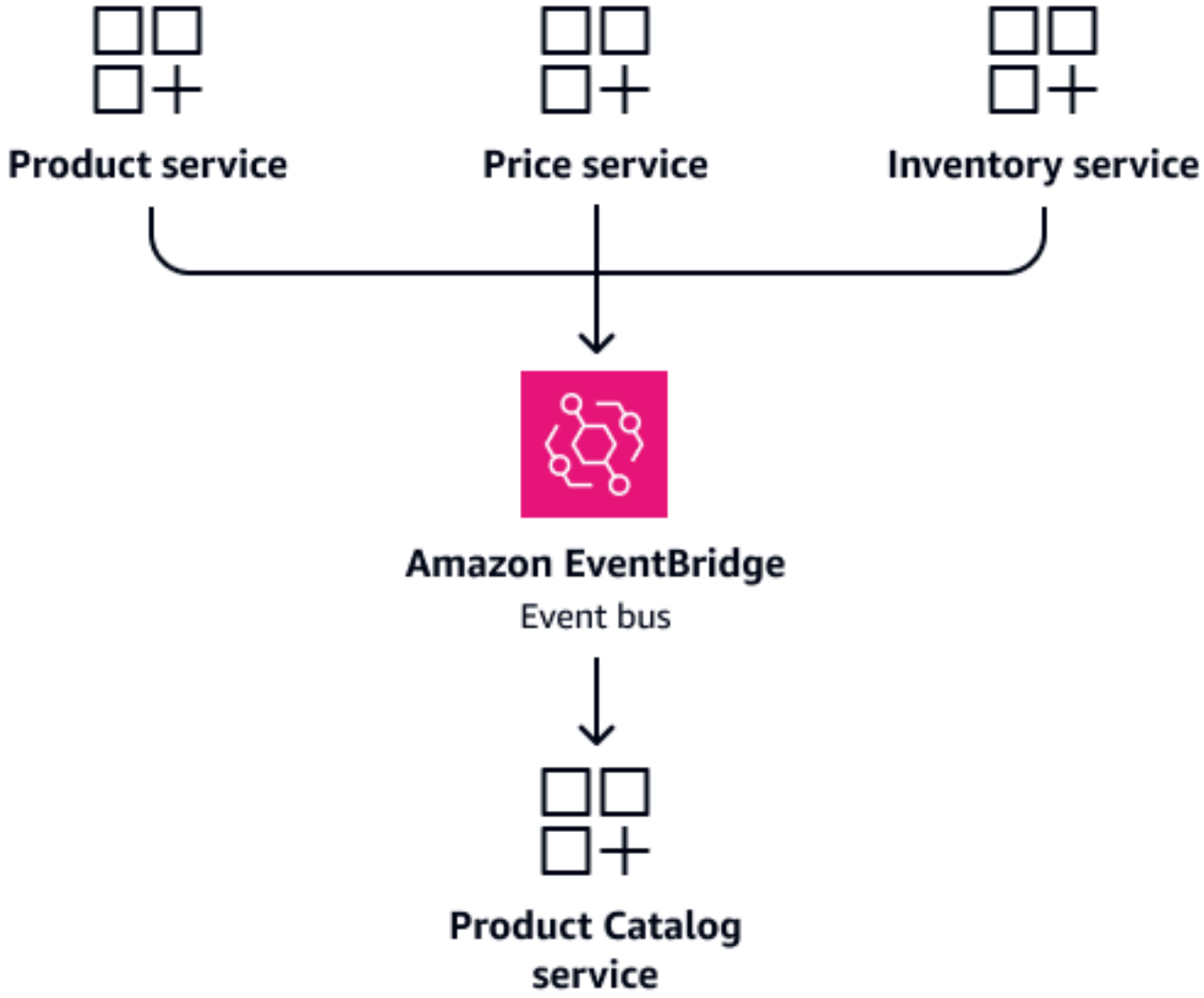
Reference-by-key strategy

A *reference-by-key strategy* is a database design pattern where relationships between entities are maintained through unique keys rather than storing the actual related data. Instead of traditional foreign key relationships, modern microservices often store just the unique identifiers of related data. For example, rather than keeping all customer details in the order table, the order service only stores the customer ID and retrieves additional customer information through an API call when needed. This approach maintains service independence while ensuring access to related data.

CQRS pattern

The *Command Query Responsibility Segregation (CQRS)* pattern separates the read and write operations of a data store. This pattern is particularly useful in complex systems with high-performance requirements, especially those with asymmetric read/write loads. If your application frequently needs data combined from multiple sources, you can create a dedicated CQRS model instead of complex joins. For example, rather than joining `Product`, `Pricing`, and `Inventory` tables on every request, maintain a consolidated `Product Catalog` table that contains the necessary data. The benefits of this approach can outweigh the costs of the additional table.

Consider a scenario where `Product`, `Price`, and `Inventory` services frequently need product information. Instead of configuring these services to directly access shared tables, create a dedicated `Product Catalog` service. This service maintains its own database that contains the consolidated product information. It acts as a single source of truth for product-related queries. When product details, prices, or inventory levels change, respective services can publish events to update the `Product Catalog` service. This provides data consistency while maintaining service independence. The following image shows this configuration, where Amazon EventBridge serves as an event bus.



As discussed in Event-based data synchronization, the next section, keep the CQRS model updated through events. When product details, prices, or inventory levels change, the respective services publish events. The `Product Catalog` service subscribes to these events and updates its consolidated view. This provides fast reads without complex joins, and it maintains service independence.

Event-based data synchronization

Event-based data synchronization is a pattern where changes to data are captured and propagated as events, which enables different systems or components to maintain synchronized data states. When data changes, instead of updating all related databases immediately, publish an event to notify subscribed services. For example, when a customer changes their shipping address in

the `Customer` service, a `CustomerUpdated` event initiates updates to the `Order` service and `Delivery` service on each service's schedule. This approach replaces rigid table joins with flexible, scalable event-driven updates. Some services might briefly have outdated data, but the trade-off is improved system scalability and service independence.

Implementing alternatives to table joins

Begin your database decomposition with read operations because they're typically simpler to migrate and validate. After read paths are stable, tackle the more complex write operations. For critical, high-performance requirements, consider implementing the CQRS pattern. Use a separate, optimized database for reads while maintaining another for writes.

Build resilient systems by adding retry logic for cross-service calls and implementing appropriate caching layers. Monitor service interactions closely, and set up alerts for data consistency issues. The end goal isn't perfect consistency everywhere—it's creating independent services that perform well while maintaining acceptable data accuracy for your business needs.

The decoupled nature of microservices introduces the following new complexities in data management:

- Data is distributed. Data now resides in separate databases, which are managed by independent services.
- Real-time synchronization across services is often impractical, necessitating an eventual consistency model.
- Operations that previously occurred within a single database transaction now span multiple services.

To address these challenges, do the following:

- **Implement an event-driven architecture** – Use message queues and event publishing to propagate data changes across services. For more information, see Building Event Driven Architectures on Serverless Land.
- **Adopt the saga orchestration pattern** – This pattern helps you manage distributed transactions and maintain data integrity across services. For more information, see Building a serverless distributed application using a saga orchestration pattern on AWS Blogs.
- **Design for failure** – Incorporate retry mechanisms, circuit breakers, and compensating transactions to handle network issues or service failures.

- **Use version stamping** – Track data versions to manage conflicts and make sure that the most recent updates are applied.
- **Regular reconciliation** – Implement periodic data synchronization processes to catch and correct any inconsistencies.

Scenario-based example

The following schema example has two tables, a `Customer` table and an `Order` table:

```
-- Customer table
CREATE TABLE customer (
    customer_id INT PRIMARY KEY,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    email VARCHAR(255),
    phone VARCHAR(20),
    address TEXT,
    created_at TIMESTAMP
);

-- Order table
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(50),
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

The following is an example of how you could use a denormalized approach:

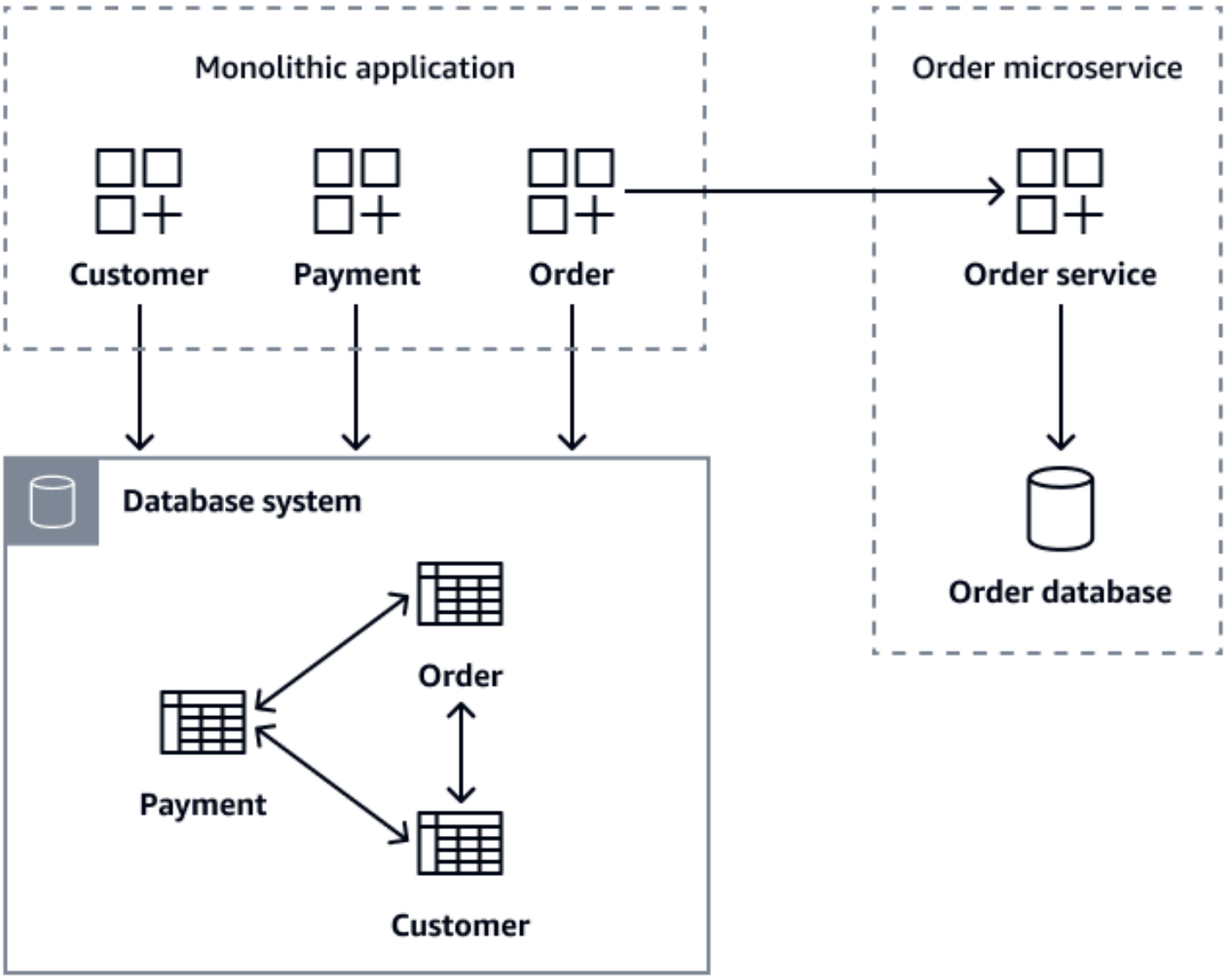
```
CREATE TABLE order (
    order_id INT PRIMARY KEY,
    customer_id INT,                  -- Reference only
    customer_first_name VARCHAR(100), -- Denormalized
    customer_last_name VARCHAR(100),  -- Denormalized
    customer_email VARCHAR(255),      -- Denormalized
    order_date TIMESTAMP,
    total_amount DECIMAL(10,2),
    status VARCHAR(50)
```

```
);
```

The new `Order` table has customer name and email addresses that are denormalized. The `customer_id` is referenced, and there is no foreign key constraint with the `Customer` table. The following are the benefits of this denormalized approach:

- The `Order` service can display order history with customer details, and it doesn't require API calls to the `Customer` microservice.
- If the `Customer` service is down, the `Order` service remains fully functional.
- Queries for order processing and reporting run faster.

The following diagram shows a monolithic application that retrieves order data using `getOrder(customer_id)`, `getOrder(order_id)`, `getCustomerOders(customer_id)`, and `createOrder(Order order)` API calls to the `Order` microservice.



During the microservices migration, you can maintain the `Order` table in the monolithic database as a transitional safety measure, ensuring that the legacy application remains functional. However, it's crucial that all new order-related operations are routed through the `Order` microservice API, which maintains its own database while simultaneously writing to the legacy database as a backup. This dual-write pattern provides a safety net. It allows for gradual migration while maintaining system stability. After all customers have successfully migrated to the new microservice, you can deprecate the legacy `Order` table in the monolithic database. After decomposing the monolithic application and its database into separate `Customer` and `Order` microservices, maintaining data consistency becomes the primary challenge.

Best practices for database decomposition

When decomposing a monolithic database, organizations must establish clear frameworks for tracking progress, maintaining system knowledge, and addressing emerging challenges. This section provides best practices for measuring decomposition success, maintaining crucial documentation, implementing continuous improvement processes, and navigating common challenges. Understanding and following these guidelines helps you make sure that database decomposition efforts deliver their intended benefits while minimizing operational disruptions and technical debt.

This section contains the following topics:

Measuring success

Track decomposition success through a mix of technical, operational, and business metrics. Technically, monitor query response times, system uptime improvements, and deployment frequency increases. Operationally, measure incident reductions, issue resolution speed, and resource utilization improvements. For development, track feature implementation speed, release cycle acceleration, and reduction in cross-team dependencies. Business impacts should result in reduced operational costs, faster time-to-market, and improved customer satisfaction. These metrics are often defined during the scope phase. For more information, see Defining the scope and requirements for database decomposition in this guide.

Documentation requirements

Maintain up-to-date system architecture documentation with clear service boundaries, data flows, and interface specifications. Use architecture decision records (ADRs) to capture key technical decisions, including their context, consequences, and alternatives considered. For example, document why specific services were separated first or how certain data consistency trade-offs were made.

Schedule monthly architecture reviews to assess system health through key metrics: performance trends, security compliance, and cross-service dependencies. Include feedback from development teams about integration challenges and operational issues. This regular review cycle helps you identify emerging problems early and validates that decomposition efforts remain aligned with business goals.

Continuous improvement strategy

Treat database decomposition as an iterative process, not a one-time project. Monitor system performance metrics and service interactions to identify optimization opportunities. Each quarter, prioritize addressing technical debt based on operational impact and maintenance costs. For example, automate frequently performed database operations, enhance monitoring coverage, and refine deployment procedures based on learned patterns.

Overcoming common challenges in database decomposition

Performance optimization requires a multi-faceted approach. Implement strategic caching at service boundaries, optimize query patterns based on actual usage, and continuously monitor key metrics. Address performance bottlenecks proactively by analyzing trends and setting clear thresholds for intervention.

Data consistency challenges demand careful architectural choices. Implement event-driven patterns for cross-service updates and use saga orchestration patterns for complex transactions. Define clear service boundaries, and accept eventual consistency where business requirements permit. This balance between consistency and service autonomy is crucial for successful decomposition.

Operational excellence requires automation of routine tasks and standardized procedures across services. Maintain comprehensive monitoring with clear alerting thresholds, and invest in regular team training for new patterns and tools. This systematic approach to operations promotes reliable service delivery while managing complexity.

FAQ for database decomposition

This comprehensive FAQ section addresses the most common questions and challenges organizations face when undertaking database decomposition projects. From defining the initial scope and requirements to migrating stored procedures, these questions provide practical insights and strategic approaches to help teams successfully navigate their database modernization journey. Whether you're in the planning phase or already executing your decomposition strategy, these answers can help you avoid common pitfalls and implement best practices for optimal results.

This section contains the following topics:

FAQs about defining scope and requirements

The Defining the scope and requirements for database decomposition section of this guide discusses how to analyze interactions, map dependencies, and establish success criteria. This FAQ section addresses key questions about establishing and managing project boundaries. Whether you're dealing with unclear technical constraints, conflicting departmental needs, or evolving business requirements, these FAQs provide practical guidance on maintaining a balanced approach.

This section contains the following questions:

How detailed should the initial scope definition be?

Working backwards from your customers' needs, define project scope with enough detail to identify system boundaries and critical dependencies while maintaining flexibility for discovery. Map essential elements, including system interfaces, key stakeholders, and major technical constraints. Start small by selecting a bounded, low-risk portion of the system that provides measurable value. This approach helps teams to learn and adjust strategies before tackling more complex components.

Document critical business requirements that drive the decomposition effort, but avoid over-specifying details that might change during implementation. This balanced approach makes sure that teams can move forward with clarity while remaining adaptable to new insights and challenges that emerge during the modernization journey.

What if I discover additional dependencies after starting the project?

Expect to uncover additional dependencies as the project progresses. Maintain a live dependency log and conduct regular scope reviews to assess impact on timelines and resources. Implement a clear change management process, and include buffer time in project plans to handle unexpected discoveries. The goal isn't to prevent changes but to manage them effectively. This helps teams to adapt quickly while maintaining project momentum.

How do I handle stakeholders from different departments who have conflicting requirements?

Handle conflicting departmental requirements through clear prioritization that is based on business value and system impact. Secure executive sponsorship to drive key decisions and resolve conflicts quickly. Schedule regular stakeholder alignment meetings to discuss trade-offs and maintain transparency. Document all decisions and their rationale to promote clear communication and maintain project momentum. Focus discussions on quantifiable business benefits rather than departmental preferences.

What's the best way to assess technical constraints when documentation is poor or outdated?

When facing poor documentation, combine traditional analysis with modern AI tools. Use large language models (LLMs) to analyze code repositories, logs, and existing documentation in order to identify patterns and potential constraints. Interview experienced developers and database

architects to validate AI findings and uncover undocumented constraints. Deploy monitoring tools that have enhanced AI capabilities in order to observe system behavior and predict potential issues.

Create small technical experiments that validate your assumptions. You can use AI-powered testing tools to accelerate the process. Document findings in a knowledge base that can be continuously enhanced through AI-assisted updates. Consider engaging subject matter experts for complex areas, and use AI pair-programming tools to accelerate their analysis and documentation efforts.

How do I balance immediate business needs with long-term technical goals?

Create a phased project roadmap that aligns immediate business needs with long-term technical objectives. Identify quick wins that deliver tangible value early so that you can build stakeholder confidence. Break down the decomposition into clear milestones. Each should deliver measurable business benefits while progressing toward architectural goals. Maintain flexibility to address urgent business needs through regular roadmap reviews and adjustments.

How do I make sure that I'm not missing critical requirements from silent stakeholders?

Map all potential stakeholders across the organization, including downstream system owners and indirect users. Create multiple feedback channels through structured interviews, workshops, and regular review sessions. Build proof-of-concepts and prototypes to make requirements tangible and spark meaningful discussions. For example, a simple dashboard that shows system dependencies often reveals hidden stakeholders and requirements that weren't initially apparent.

Conduct regular validation sessions with both vocal and quiet stakeholders, and make sure that all perspectives are captured. Critical insights often come from those closest to daily operations rather than the loudest voices in the planning meetings.

Do these recommendations apply for monolithic mainframe databases?

The methodology described in this guide also applies to decomposing monolithic mainframe databases. The primary challenges with these databases are managing requirements from the various stakeholders. The technology recommendations in this guide might apply to monolithic mainframe databases. If the mainframe has a relational database, such as an online transaction processing (OLTP) database, then many of the recommendations apply. For online analytical processing (OLAP) databases, such as those used to generate business reports, then only some of the recommendations apply.

FAQs about controlling database access

Controlling database access by using the database wrapper service pattern is discussed in the Controlling database access during decomposition section of this guide. This FAQ section addresses common concerns and questions about introducing a database wrapper service, including its potential impact on performance, handling of existing stored procedures, managing complex transactions, and overseeing schema changes.

This section contains the following questions:

Won't the wrapper service become a new bottleneck?

While the database wrapper service does add an extra network hop, the impact is usually minimal. You can scale the service horizontally, and the benefits of controlled access typically outweigh the small performance cost. Consider it a temporary trade-off between performance and maintainability.

What happens to existing stored procedures?

Initially, the database wrapper service can expose stored procedures as service methods. Over time, you can gradually move the logic into the application layer, which improves testing and version control. Migrate the business logic incrementally to minimize risk.

How do I manage schema changes during the transition?

Centralize schema change control through the wrapper service team. This team is responsible for maintaining comprehensive visibility across all consumers. This team reviews proposed changes for system-wide impact, coordinates with affected teams, and implements modifications by using a controlled deployment process. For instance, when adding new fields, this team should maintain backward compatibility by implementing default values or initially allowing nulls.

Establish a clear change management process that includes impact assessment, testing requirements, and rollback procedures. Use database versioning tools, and maintain clear

documentation of all changes. This centralized approach prevents schema modifications from disrupting dependent services and maintains system stability.

FAQs about analyzing cohesion and coupling

Understanding and effectively analyzing database coupling and cohesion is fundamental to successful database decomposition. Coupling and cohesion are discussed in the Analyzing cohesion and coupling for database decomposition section of this guide. This FAQ section addresses key questions about identifying appropriate levels of granularity, selecting the right analysis tools, documenting findings, and prioritizing coupling issues.

This section contains the following questions:

How do I identify the right level of granularity when analyzing coupling?

Start with a broad analysis of database relationships, then systematically drill down to identify natural separation points. Use database analysis tools to map table-level relationships, schema dependencies, and transaction boundaries. For example, examine join patterns in SQL queries to understand data access dependencies. You can also analyze transaction logs to identify business process boundaries.

Focus on areas where coupling is naturally minimal. These often align with business domain boundaries and represent optimal decomposition points. When determining appropriate service boundaries, consider both technical coupling (such as shared tables and foreign keys) and business coupling (such as process flows and reporting needs).

What tools can I use to analyze database coupling and cohesion?

You can use a combination of automated tools and manual analysis to assess database coupling and cohesion. The following tools can help you with this assessment:

- **Schema visualization tools** – You can use tools like SchemaSpy or pgAdmin to generate ER diagrams. These diagrams reveal table relationships and potential coupling points.
- **Query analysis tools** – You can use pg_stat_statements or SQL Server Query Store to identify frequently joined tables and access patterns.
- **Database profiling tools** – Tools such as Oracle SQL Developer or MySQL Workbench provide insights into query performance and data dependencies.
- **Dependency mapping tools** – The AWS Schema Conversion Tool (AWS SCT) can help you visualize schema relationships and identify tightly coupled components. vFunction can help you identify domain boundaries by analyzing the application's functional and domain boundaries.
- **Transaction monitoring tools** – You can use database-specific tools, such as Oracle Enterprise Manager or SQL Server Extended Events, to analyze transaction boundaries.
- **Business logic migration tools** – You can use Ispirer or generative AI tools, such as Amazon Q Developer or Kiro, to convert database business logic for the application layer, such as conversion to Java.

Combine these automated analyses with manual review of business processes and domain knowledge to fully understand system coupling. This multi-faceted approach makes sure that both technical and business perspectives are considered in your decomposition strategy.

What's the best way to document coupling and cohesion findings?

Create comprehensive documentation that visualizes database relationships and usage patterns. The following are the types of assets that you can use to record your findings:

- **Dependency matrices** – Map table dependencies and highlight high-coupling areas.
- **Relationship diagrams** – Use ER diagrams to show schema connections and foreign key relationships.
- **Table usage heat maps** – Visualize query frequency and data access patterns across tables.
- **Transaction flow diagrams** – Document multi-table transactions and their boundaries.
- **Domain boundary maps** – Outline potential service boundaries based on business domains.

Combine these artifacts in a document, and regularly update it as the decomposition progresses. For diagrams, you can use tools such as draw.io or Lucidchart. Consider implementing a wiki for easy team access and collaboration. This multi-faceted documentation approach provides a clear, shared understanding of system coupling and cohesion.

How do I prioritize which coupling issues to address first?

Prioritize coupling issues based on a balanced assessment of business and technical factors. Evaluate each issue against business impact (such as revenue and customer experience), technical risk (such as system stability and data integrity), implementation effort, and team capabilities. Create a prioritization matrix that scores each issue from 1-5 across these dimensions. This matrix helps you identify the most valuable opportunities with manageable risks.

Start with high-impact, low-risk changes that align with existing team expertise. This helps you build organizational confidence and momentum for more complex changes. This approach promotes realistic execution and maximizes business value. Regularly review and adjust the priorities to help maintain alignment with changing business needs and team capacity.

How do I handle transactions that span multiple operations?

Handle multi-operation transactions through carefully designed service-level coordination. Implement saga patterns for complex distributed transactions. Break them into smaller, reversible steps that can be managed independently. For example, an order processing flow might be split into separate steps for inventory check, payment processing, and order creation, each with its own compensation mechanism.

Where possible, redesign operations to be more atomic, which reduces the need for distributed transactions. When distributed transactions are unavoidable, implement robust tracking and compensation mechanisms to promote data consistency. Monitor transaction completion rates and implement clear error recovery procedures to maintain system reliability.

FAQs about migrating the business logic to the application layer

Migrating business logic from the database to the application layer is a critical and complex aspect of database modernization. This business logic migration is discussed in the Migrating business logic from the database to the application layer section of this guide. This FAQ section addresses common questions about managing this transition effectively, from selecting initial candidates for migration to handling complex stored procedures and triggers.

This section contains the following questions:

How do I identify which stored procedures to migrate first?

Start by identifying stored procedures that offer the best combination of low-risk and high-learning value. Focus on procedures that have minimal dependencies, clear functionality, and non-critical business impact. These make ideal candidates for initial migration because they help the team build confidence and establish patterns. For example, choose procedures that handle simple data operations over those that manage complex transactions or critical business logic.

Use database monitoring tools to analyze usage patterns and identify infrequently accessed procedures as early candidates. This approach minimizes business risk while providing valuable experience for tackling more complex migrations later. Score each procedure on complexity, business criticality, and dependency levels to create a prioritized migration sequence.

What are the risks of moving logic to the application layer?

Moving database logic to the application layer introduces several key challenges. System performance can degrade due to increased network calls, especially for data-intensive operations that were previously handled within the database. Transaction management becomes more complex and requires careful coordination to maintain data integrity across distributed operations. Ensuring data consistency becomes challenging, particularly for operations that previously relied on database-level constraints.

Potential business disruption during the migration and the learning curve for developers are also significant concerns. Mitigate these risks through thorough planning, extensive testing in staged environments, and gradual migration that starts with less-critical components. Implement robust monitoring and rollback procedures to quickly identify and address issues in production.

How do I maintain performance when moving logic away from the database?

Implement appropriate caching mechanisms for frequently accessed data, optimize data access patterns to minimize network calls, and use batch processing for bulk operations. For non-time-critical operations, consider asynchronous processing to improve system responsiveness.

Monitor application performance metrics closely and tune them as needed. For example, you can replace multiple single-row operations with bulk processing, you can cache reference data that changes infrequently, and you can optimize query patterns to reduce data transfer. Regular performance testing and tuning helps the system maintain acceptable response times and improves maintainability and scalability.

What should I do with complex stored procedures that involve multiple tables?

Approach complex, multi-table stored procedures through systematic decomposition. Start by breaking them into smaller, logically coherent components, and identify clear transaction boundaries and data dependencies. Create service interfaces for each logical component. This helps you gradually migrate without disrupting the existing functionality.

Implement a step-by-step migration, starting with the least coupled components. For highly intricate procedures, consider temporarily keeping them in the database while migrating simpler parts. This hybrid approach maintains system stability while you progress toward your architectural goals. Continuously monitor performance and functionality during the migration, and be prepared to adjust your strategy based on the results.

How do I handle database triggers during migration?

Transform database triggers into application-level event handlers while maintaining system functionality. Replace synchronous triggers with event-driven patterns that message queues for asynchronous operations. Consider using Amazon Simple Notification Service (Amazon SNS) or Amazon Simple Queue Service (Amazon SQS) for the message queues. For audit requirements, implement application-level logging or use database change data capture (CDC) features.

Analyze each trigger's purpose and criticality. Some triggers might be better served by application logic, and others might require event-sourcing patterns to maintain data consistency. Start with

simple triggers, such as audit logs, before tackling complex ones that manage business rules or data integrity. Monitor carefully during the migration to make sure that there is no loss of functionality or data consistency.

What's the best way to test the migrated business logic?

Implement a multi-layered testing approach before you deploy the migrated business logic. Start with unit tests for new application code, then add integration tests that cover end-to-end business flows. Run old and new implementations in parallel, and then compare the results in order to validate functional equivalence. Conduct performance testing under various load conditions to verify that the system behavior matches or exceeds previous capabilities.

Use feature flags to control deployment so that you can quickly roll back if issues arise. Involve business users in the validation, particularly for critical workflows. Monitor key metrics during initial deployment, and gradually increase traffic to the new implementation. Throughout, maintain the ability to revert to the original database logic if needed.

How do I manage the transition period when both database and application logic exist?

When the database and application logic are both in use, implement feature flags that control traffic flow and enable quick switching between old and new implementations. Maintain rigorous version control, and clearly document both implementations and their respective responsibilities. Set up comprehensive monitoring for both systems to quickly identify any discrepancies or performance issues.

Establish clear rollback procedures for each migrated component so that you can revert to the original logic if needed. Communicate regularly with all stakeholders about the transition status, potential impacts, and escalation procedures. This approach helps you gradually migrate while maintaining system stability and stakeholder confidence.

How do I handle error scenarios in the application layer that were previously managed by the database?

Replace database-level error handling with robust application-layer mechanisms. Implement circuit breakers and retry logic for transient failures. Use compensating transactions for maintaining data consistency across distributed operations. For example, if a payment update fails, the application should automatically retry within defined limits and initiate compensating actions if needed.

Set up comprehensive monitoring and alerting to quickly identify issues, and maintain detailed audit logs for troubleshooting. Design error handling to be as automated as possible, and define clear escalation paths for scenarios that require human intervention. This multi-layered approach provides system resilience while maintaining data integrity and business process continuity.

Next steps for database decomposition on AWS

After implementing initial database decomposition strategies through database wrapper services and moving business logic to the application layer, organizations must plan their next evolution. This section outlines key considerations for continuing your modernization journey.

This section contains the following topics:

Incremental strategies for database decomposition

Database decomposition follows a gradual evolution through three distinct phases. Teams first wrap the monolithic database with a database wrapper service to control access. They then begin splitting the data into service-specific databases, while maintaining the primary database for legacy needs. Finally, they complete migrate the business logic in order to transition to fully independent service databases.

Throughout this journey, teams must implement careful data synchronization patterns and continuously validate consistency across services. Performance monitoring becomes crucial to identify and address potential issues early. As services evolve independently, their schemas should be optimized based on actual usage patterns, and you should remove redundant structures that accumulated over time.

This incremental approach helps minimize risks while maintaining system stability throughout the transformation process.

Technical considerations for distributed database environments

In a distributed database environment, performance monitoring becomes essential to identify and address bottlenecks early. Teams must implement comprehensive monitoring systems and caching strategies to maintain performance levels. Read/write splitting can effectively balance loads across the system.

Data consistency requires careful orchestration across distributed services. Teams should implement eventual consistency patterns where appropriate and establish clear data ownership boundaries. Robust monitoring promotes data integrity across all services.

In addition, security must evolve to accommodate the distributed architecture. Each service needs fine-grained security controls, and your access patterns require regular review. Enhanced monitoring and auditing become critical in this distributed environment.

Organizational changes to support distributed architectures

The team structure should align with service boundaries in order to define clear ownership and accountability. Organizations must establish new communication patterns and build additional technical capabilities within teams. This structure should support both maintenance of existing services and your continued architectural evolution.

You must update your operational processes to handle the distributed architecture. Teams must modify deployment procedures, adapt incident response processes, and evolve change management practices to coordinate across multiple services.

Resources

The following additional resources and tools can help your organization on its database decomposition journey.

AWS Prescriptive Guidance

- Migrating Oracle databases to the AWS Cloud
- Replatform options for Oracle Database on AWS
- Cloud design patterns, architectures, and implementations

AWS blog posts

- Migrate business logic from database to application for faster innovation and flexibility

AWS services

- AWS Application Migration Service
- AWS Database Migration Service (AWS DMS)
- Migration Evaluator
- AWS Schema Conversion Tool (AWS SCT)
- AWS Transform

Other tools

- AppEngine (Dynatrace website)
- Oracle Automatic Workload Repository (Oracle website)
- CAST Imaging (CAST website)
- Kiro (Kiro website)
- pgAdmin (pgAdmin website)
- pg_stat_statements (PostgreSQL website)
- SchemaSpy (SchemaSpy website)

- SQL Developer (Oracle website)
- SQLWays (Ispirer website)
- vFunction (vFunction website)

Other resources

- Monolith to microservices (O'Reilly website)

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
Mainframe FAQ and AI tools	We added the Do these recommendations apply for monolithic mainframe databases? FAQ, and we added additional informati on about AI tools that you can use during database decomposition.	October 14, 2025
Initial publication	—	September 30, 2025

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

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.

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.

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.

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.

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.

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.

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.

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.

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. For more information, see the Quantifying uncertainty in deep learning systems guide.

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.