What does software integration mean?

Definition

Software integration is a development process in which separate software systems—applications and components—are connected so they work together to form a new, unified system.

Integration activity phases

1.) Requirements assessment and planning

- requirements assessment: identifying user needs and business requirements
- planning: developing the integration strategy, designing the system architecture and integration points

2.) Requirements analysis and specification

- analysis: detailed assessment of required functions and the capabilities of existing systems
- specification: definition of interfaces and data exchange formats

3.) Development and implementation

- development: creation of integration code for existing systems
- implementation: deployment of new components and modification of existing ones

4.) Testing and validation

5.) Maintenance and support

Legacy Systems

Definition A legacy system is an IT system that uses older (possibly obsolete) technologies but is still actively operating and plays an essential role in an organisation's everyday operations.

Why do we still use legacy systems?

- Long lifetime and stability: Many legacy systems have operated reliably for years or even decades. If a system functions well and is mission-critical, there is often no compelling reason to replace it.
- Cost considerations: replacing an entire system can be extremely expensive.
- Complexity: legacy systems are often deeply integrated into other organizational processes and systems.
- Risk avoidance: organizations often avoid the risk of replacement

Why are they not replaced?

- Cost and lack of resources: replacement is often not economically feasible, and risk estimation can be difficult due to strict security requirements.

- Disruption of business processes: introducing a new system may significantly disrupt daily operations. Many organizations prioritize operational stability (e.g., banking institutions).

Solutions

- Gradual migration: instead of replacing the entire system at once, organizations incrementally migrate to new systems by replacing specific components or functionalities.
- Outsourcing maintenance and operation: maintenance and operation of legacy systems may be delegated to external service providers, reducing internal costs and specialized staffing requirements.

Interesting Real-World examples of Legacy Systems

In many cases, legacy systems serve as the backbone of critical global infrastructure. Some of the most striking examples can be found in the financial sector. Major institutions such as *JPMorgan Chase* and *Bank of America* still rely heavily on COBOL-based core banking systems running on IBM mainframes. COBOL, introduced in 1959, remains responsible for processing an enormous portion of the world's financial transactions. During the COVID-19 pandemic, the continued reliance on COBOL became apparent to the broader public when U.S. government agencies urgently sought experienced COBOL developers to maintain unemployment benefit systems.

Another remarkable case is the airline reservation infrastructure originally developed for American Airlines in cooperation with IBM in the early 1960s. The SABRE system was one of the earliest large-scale real-time transaction processing systems. Although it has undergone continuous modernization, elements of its original architecture still influence its operation today. The system handles millions of transactions daily and demonstrates how evolutionary development can preserve legacy foundations while incrementally adapting to new technological requirements.

Public administration also provides compelling examples. The U.S. Internal Revenue Service (IRS) continues to operate tax processing systems that date back to the 1960s. Despite multiple large-scale modernization programs costing billions of dollars, complete replacement has proven extremely complex due to regulatory constraints, security requirements, and the critical importance of uninterrupted service.

In transportation infrastructure, legacy technology can even affect physical operations. Parts of the London Underground's signaling and control systems have historically relied on hardware and software developed decades ago. In some cases, replacement components had to be sourced from secondary markets because original manufacturers no longer produced them. Such situations illustrate not only software legacy issues but also hardware obsolescence and supply chain risks.

Overview of Integration Strategies

Point to Point connection

Components connect directly to each other, typically via file transfer or direct database access. There

is no intermediary layer, therefore communication is fast. Initially, it is easy to implement.

flowchart LR %% Nodes R[Radiology] EMR[EMR] CDB[Central Database] PS[Patient Search] PDB[Patient DB] ER[Emergency Dept.] FIN[Billing / Finance] PHARM[Pharmacy] %% Layout helpers (optional) %% Try to mimic the original positions by grouping subgraph Left[ ] direction TB R PS FIN end subgraph Middle[ ] direction TB EMR PDB PHARM end subgraph Right[ ] direction TB CDB ER end %% Connections (based on the diagram) R <--> PS PS <--> EMR PS --> PDB PS --> FIN EMR --> PDB EMR --> FIN EMR <--> CDB PDB <--> ER PDB <--> PHARM ER --> PHARM PHARM --> ER CDB --> ER ER --> CDB

Disadvantages – Challenges

- Difficult to scale
- Future expansion can become complex
- The number of connections grows exponentially → n(n−1)/2 connections
- Fragile architecture, as monitoring and troubleshooting errors are difficult

---

Middleware Integration

Components do not connect to each other directly; instead, they communicate through a central intermediary (e.g., API Gateway, Application Server, Enterprise Service Bus – ESB).

The intermediary layer handles different communication protocols.

- Monitoring capabilities: communication tracking and centralized supervision
- Improved scalability
- Centralized functions: authorization management and transaction handling

Disadvantages – Challenges

- Complexity: system development costs are high
- Monolithic architecture – one central server serving many clients

flowchart TB EMR[EMR] RAD[Radiology] FIN[Billing and Finance] ER[Emergency Department] MW[Middleware] PS[Patient Search] PDB[Patient Database] CDB[Central Database] PHARM[Pharmacy] EMR --> MW RAD --> MW FIN --> MW ER --> MW MW --> PS MW --> PDB MW --> CDB MW --> PHARM

Message Queue-Based Integration

Components do not connect to each other directly; instead, they communicate via message queues.

Messages are processed asynchronously.

- Monitoring capabilities: use of specialized queues (e.g., Dead Letter Queue – DLQ)

- Highly scalable architecture
- Naturally suited for cloud-based systems

Disadvantages – Challenges

- Complexity: system development costs are high
- Requires advanced expertise and careful architectural design

flowchart TB %% Top layer systems RAD[Radiology] EMR[EMR] FIN[Billing] ER[Emergency Department] %% Message Queues Q1[[Queue]] Q2[[Queue]] Q3[[Queue]] %% Bottom layer systems PS[Patient Search] PDB[Patient Database] CDB[Central Database] PHARM[Pharmacy] %% Top -> Queues RAD --> Q1 EMR --> Q1 FIN --> Q2 ER --> Q3 %% Queue interconnection Q2 <--> Q3 %% Queues -> Bottom Q1 --> PS Q2 --> PDB Q3 --> CDB Q3 --> PHARM

Data Sharing

A simple approach to integration is data sharing. Data sharing–based integration aims to transfer and share data between systems. This enables individual systems to access and utilize data stored in other systems.

Data sharing can take several forms:

Comparison of Data Sharing Approaches

- **Data Migration** is typically a one-time process: It is suitable for system replacement or major upgrades, but it does not ensure continuous consistency.
- **Data Synchronization** provides ongoing consistency between systems: It is appropriate when multiple systems must maintain aligned datasets over time.
- **Data Sharing Services** enable real-time access to shared data: They are ideal for modern distributed and cloud-based architectures that require immediate data availability.

File-Based Data Sharing

The most fundamental method of data sharing. One application writes data, while another application reads data from the same file. The data files are stored in a central location — such as a shared folder (e.g., NFS) or an (S)FTP server. The information flow is unidirectional: A → B.

Data Encoding

Most file-based integration approaches use text-based files.

The most common formats are:

- Plain text
- XML, CSV, YAML
- JSON (in modern systems)

Raw text formats may use:

- Fixed-length records
- Variable-length records (commonly used in billing and financial systems)

For variable-length records, a delimiter is required to separate data fields. The most widely known method is CSV (Comma-Separated Values).

flowchart LR A[System A] STORAGE[[Shared Folder\nFTP Server]] B[System B] A -- writes --> STORAGE STORAGE -- reads --> B

File-Based Integration with Lock Mechanism

State Files

State files can be used to track the processing status of data files.

These files may contain the current processing state, such as:

- “in progress”
- “completed”
- “failed”

File-Based Integration with Lock Mechanism

State Files

State files can be used to track the processing status of data files.

These files may contain the current processing state, such as:

- “in progress”
- “completed”
- “failed”

flowchart LR A[System A] STORAGE[[Shared Folder / FTP Server]] LOCK[(data.lock)] B[System B] A -- "1) create lock" --> LOCK A -- "2) write data" --> STORAGE B -- "3) detect lock" --> LOCK B -- "waits" --> STORAGE A -- "4) remove lock" --> LOCK B -- "5) read data" --> STORAGE

Lock File Mechanism

1) Lock file creation: System A begins processing a data file and creates a lock file, for example:

data.lock.

2) Writing phase: System A creates or writes the data file while *data.lock* exists. System B attempts to access the data file but detects that the lock file exists, therefore it waits.

3) Completion: System A finishes processing and removes the data.lock file.

4) Reading phase: System B detects the *data.lock* file has been removed, and it can begin its own processing.

Purpose of the Lock Mechanism

This method ensures that only one system processes the data file at a time, preventing data conflicts and inconsistencies.

The use of lock files is a simple and effective technique for process synchronization and coordination in file-based integration.

Limitations of File-Based Integration

This method remains widely used today, but it has several significant disadvantages:

- The data sharing is not real-time. It is typically suitable for daily, weekly, or monthly batch data exchange. If data is modified between cycles — for example, if a customer changes their address — the invoicing application may still send the invoice to the old address because it receives the update only later.
- It may become unreliable when transferring a large number of files (although tools such as rsync can help).
- Successful integration requires that developers of both applications (in most cases) agree on and understand:
    1. the file format
    2. file naming conventions
    3. file storage location
    4. how file deletion is handled
    5. the lock mechanism used
    6. the file transfer method

Database-Based Data Sharing

Database-based integration is a method that enables data sharing and synchronization between different systems directly through databases.

In this approach, multiple applications and systems use either:

- a shared database, or
- database replication

to access and manage data.

flowchart LR A[Application A] B[Application B] DB[(Shared Database)] A <--> DB B <--> DB

The same system with db replication:

flowchart LR A[Application A] C[Application C] DB1[(Primary Database)] B1[Application B] B2[Application B] DB2[(Replica Database)] A <-- read/write --> DB1 C <-- read/write --> DB1 DB1 -- replication --> DB2 DB2 -- read --> B1 DB2 -- read --> B2

Example

E-commerce platform and Warehouse Management System: The e-commerce platform can be directly integrated with the warehouse database to provide real-time inventory information.

Similarities to File-Based Integration

- Platform-independent connectivity (e.g., JDBC, ODBC)
- Multiple instances of identical components may access the same database
    - Synchronization issue: Who processes the next record in the queue?
    - However, it can be an ideal solution for data collection scenarios.

Limitations

- Not real-time by default. If one application writes to the database, another application does not automatically receive notification.
    - Possible solutions:
        - Database notification mechanisms (e.g., PostgreSQL LISTEN and NOTIFY)
        - Triggers
        - Polling mechanisms
- Security considerations: Properly defined access rights are required (table access, permitted operations). Developers often restrict direct visibility of database structures.
- Lack of well-defined interfaces — this approach provides only data-level integration.

When to Use Database-Based Integration?

Database-based integration is appropriate in the following scenarios:

- When multiple applications need direct access to the same structured data.
- When strong transactional consistency is required.
- When systems operate within the same organizational or security boundary.
- When the data model is stable and well-defined.
- When high-performance querying and reporting are necessary.

- When database replication can support load balancing or read scalability.

However, it may NOT be the best choice:

- In loosely coupled, distributed architectures (e.g., microservices).
- When clear service-level interfaces are required.
- When strict decoupling between systems is a design goal.
- In highly scalable cloud-native environments where message-based communication is preferred.

Integration Strategy Comparison

Aspect	**File-Based Integration**	**Database-Based Integration**	**Message Queue-Based Integration**
Coupling	Tight coupling (shared file format)	Tight to medium coupling (shared schema)	Loose coupling
Communication Style	Batch, unidirectional	Data-level sharing	Asynchronous message exchange
Real-Time Capability	No	Not by default	Yes (naturally asynchronous)
Scalability	Limited	Moderate	High
Monitoring	Difficult	Database-level monitoring	Built-in queue monitoring (DLQ, metrics)
Complexity	Low initial complexity	Medium	High
Transaction Support	No native support	Strong ACID support	Depends on message broker
Typical Use Case	Periodic data exchange	Shared enterprise systems	Distributed / cloud-native systems
Interface Definition	File format agreement	Shared database schema	Message contract / schema definition
Cloud-Native Suitability	Low	Medium	High