IT Project Guidance

Integrations – Overview of Common Approaches

Version:

0.2

## Purpose

This document supports system design decisions by outlining and comparing common integration approaches used between digital services. It aims to provide a clear understanding of the trade-offs involved in each method, enabling both technical teams and decision-makers to assess suitability based on system capabilities, operational constraints, and strategic priorities.

## Synopsis

While Integration APIs are often preferred for their reliability and maintainability, they are not always available or feasible. Environmental constraints, legacy systems, or cost considerations may necessitate alternative integration methods. This document presents a structured comparison of these approaches to inform design choices in a consistent and transparent manner.

## Contents

[Purpose 1](#_Toc160092161)

[Synopsis 1](#_Toc160092162)

[Contents 2](#_Toc160092163)

[Background 3](#_Toc160092164)

[Integration Approaches 4](#_Toc160092165)

[User Interface (UI) based Integrations 4](#_Toc160092166)

[Hypertext Links 4](#_Toc160092167)

[Advantages 4](#_Toc160092168)

[Considerations 4](#_Toc160092169)

[Disadvantages 4](#_Toc160092170)

[Application Programming Interface (API) based Integrations 5](#_Toc160092171)

[Webhooks 5](#_Toc160092172)

[Advantages 5](#_Toc160092173)

[Considerations 5](#_Toc160092174)

[Disadvantages 6](#_Toc160092175)

[Integration APIs 6](#_Toc160092176)

[Advantages 6](#_Toc160092177)

[Considerations 7](#_Toc160092178)

[Disadvantages 7](#_Toc160092179)

[Non-Integration APIs 7](#_Toc160092180)

[Advantages 7](#_Toc160092181)

[Disadvantages 8](#_Toc160092182)

[Non Interface Based Integrations 8](#_Toc160092183)

[Advantages 9](#_Toc160092184)

[Considerations 9](#_Toc160092185)

[Disadvantage 9](#_Toc160092186)

[Conclusion 9](#_Toc160092187)

[Appendices 10](#_Toc160092188)

[Appendix A - Document Information 10](#_Toc160092189)

[Versions 10](#_Toc160092190)

[Images 10](#_Toc160092191)

[Tables 10](#_Toc160092192)

[References 10](#_Toc160092193)

[Review Distribution 10](#_Toc160092194)

[Audience 10](#_Toc160092195)

[Structure 10](#_Toc160092196)

[Diagrams 10](#_Toc160092197)

[Terms 11](#_Toc160092198)

## Summary

Integration serves as the backbone of modern digital services, enabling different systems to exchange data, coordinate processes, and provide seamless experiences for users and organisations. However, while integration and interoperability are often mentioned together, they have distinct meanings. Integration typically connects a system to specific external services in a targeted, one-way fashion, much like building a private bridge between two buildings. In contrast, interoperability is about making a system accessible to any approved external party by following agreed-upon standards, akin to creating doors with standard locks for broader access. Integration can take advantage of interoperability when open standards are available, making it easier to connect with multiple systems.

There are several approaches to system integration, each carrying different implications for cost, speed, sustainability, and security. User Interface (UI)-based integration is the simplest and most familiar method, allowing users to manually interact with multiple applications through features such as links or buttons. While quick to implement, this approach can be fragile if the underlying systems change. Automated system-to-system integration goes a step further by enabling direct digital communication between systems using APIs or messaging protocols. This method is generally more robust and efficient but requires careful alignment of data models and security measures. An alternative is indirect integration, which relies on methods like file-based data exchanges, allowing systems to share information through file transfers rather than direct connections.

Ideally, secure integration APIs using current patterns, protocols, and standards should be employed. Yet, these options are not always available, so fallback methods may be necessary. Decision makers must weigh several factors when choosing an integration method. More sophisticated integrations may offer better performance and features but can be more challenging to maintain. Automated integrations are typically more reliable than manual ones, though they demand thoughtful planning and testing. Each connection between systems introduces potential security risks, so the chosen approach must strike a balance between ease of access and protection of sensitive information. Additionally, long-term sustainability is crucial; the integration should remain functional as systems evolve and adapt to future changes.

In summary, selecting the right integration approach is not only a technical consideration but a strategic one. By understanding the differences between integration and interoperability, and assessing the respective strengths and limitations of various methods, leaders are better equipped to make informed choices that support their organisation’s digital future.

## Background

Integration between systems is a foundational aspect of digital service delivery. It enables data exchange, process coordination, and user experience continuity across organisational boundaries.

Integration and interoperability are often conflated, yet they differ substantially. Integration connects a system to specific external services—usually in a one-way, targeted manner. In contrast, interoperability focuses on designing services that are open and accessible to any external consumer following established standards, enabling broader connectivity. Ultimately, integration links to known services, whereas interoperability prepares a service to work for a wide range of others. Integration can leverage interoperable endpoints offered by other systems if they are available.

Integration methods vary significantly in terms of complexity, reliability, and maintainability. They may involve direct user interaction (e.g., through user interfaces), automated system-to-system communication (e.g., via APIs or messaging protocols), or indirect mechanisms such as file-based exchanges.

Each method carries implications for performance, security, operational overhead, and long-term sustainability. For example, user interface-based integrations may be quick to implement but fragile under change, while API-based integrations offer robustness but require alignment in data models and authentication schemes.

This document categorises integration approaches into interface-based and non-interface-based methods, and evaluates each against a consistent set of criteria: advantages, considerations, and disadvantages. The goal is to support informed decision-making that balances technical feasibility with organisational context.

# Integration Approaches

Below are listed the advantages, considerations, and disadvantages of various approaches for consideration before making integration design decisions.

## User Interface (UI) based Integrations

User Interface (UI) based integrations serve as one of the most accessible bridges between digital services, allowing users to move seamlessly from one application or platform to another. These approaches prioritise ease of use and quick implementation, relying on familiar interface elements to create connections across systems. Before delving into the specific methods, such as hypertext links, it is important to understand how UI-based integrations shape user journeys and influence the overall cohesion of service ecosystems.

### Hypertext Links

The simplest integrations between services are user activated Hypertext Links from one view within the user interface of one web service to another view, within the interface of the same or another web service.

Examples include incoming linking a project site from a corporate intranet, extranet, or product brochure site. Outgoing links may connect users from within an application or project site to external resources such as the product’s help website or a feedback service, supporting user self-service or collection of user input.

Hypertext links are also used to connect from an intranet or extranet corporate site to a project’s dedicated static information site.

From there, links are used to connect to one of the following targets:

* A public view of the solution’s system
* A protected view of the solution’s system, which redirects it to its login form,
* An IdP service separate from the target service, passing information as a query param to indicate where to return to[[1]](#footnote-2).

#### Advantages

The simplest of integration approaches, requiring no encoding of timers, triggers, etc. to as the integration is initiated by a user.

#### Considerations

Hyperlinks are capable of passing limited contextual information—such as the identifier of the current view or control within an application. For instance, when a user selects a help link, information about the specific page being viewed can be sent to the help service, enabling it to display targeted support content.

Quality assurance through automated user interface testing tools (e.g., Tosca, Selenium) is particularly challenging; tests can be brittle, difficult to maintain, and not easily integrated into automation pipelines. As a best practice, it is recommended to avoid UI automated testing when possible and instead focus on testing the application logic of the APIs used by the view. This approach ensures the functionality is tested directly, rather than UI rendering or pre-validation.

As with all channels enabling integration and interoperability, hyperlinks must be secured. It is essential to use secure HTTP (HTTPS) to protect the channel.

#### Disadvantages

The Integrations are manually triggered by users so there is no certainty of the integrations happening regularly or at all.

The link information being encoded into the client system’s interface, the information transmitted (if any at all) never changes -- the only information changed between calls is the time of the event.

## Application Programming Interface (API) based Integrations

### Webhooks

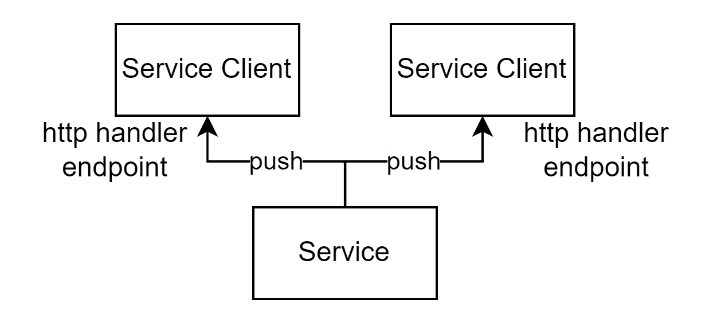


Figure 1: Web hooks based integration

With server-offered APIs, such as REST, service clients are required to continuously poll the server to check for the latest changes. This polling process can significantly increase network traffic, reducing available I/O threads and bandwidth for processing user requests, or alternatively, introduce delays when attempting to conserve bandwidth. The impact of this approach is magnified when there are large numbers of service clients, each independently making requests to the server.

Webhooks provide a solution by shifting the responsibility to the server, which actively notifies registered clients of changes by sending requests to their webhook endpoints. This eliminates the need for client-based polling. Webhooks function as one-way, publicly accessible integrations that do not require the prior establishment of a user session.

Additionally, webhook calls are asynchronous and allow the server to send notifications to clients without waiting for a response, making the integration process more efficient and streamlined.

Note:  
The approach is classifiable as a [PUSH](#Term_PUSH) pattern, as opposed to a [PULL](#Term_PULL) pattern.

#### Advantages

Webhooks[[2]](#footnote-3) remove the need for cyclical polling, decreasing unnecessary traffic and IO consumption, as well as the delay before appropriate clients are notified of changes on the server.

#### Considerations

Idempotency: Webhooks for the most part should be POST HTTP verb based as they are providing information, not retrieving information, which would require a GET. Less common, and more of an anti-pattern, Webhook handlers can be triggered from UI HTML GET based Links.

Response: Webhooks may have response payloads, but it is less common than making the webhook appropriate for a non-acknowledged “fire and forget” call.

Confidentiality: As with any connection between devices, especially across multiple networks, it should be secured using HTTP/S over TCP/IP domain-based TLS certificate.

Integrity: The validity of payloads can be verified if a secret has been provided to the service so that it can develop a cryptographic hash signature of the payload.

Dependencies: For a web-based service to hook a webhook endpoint on a client system on a private network, a reverse proxy is used.

Portability: If used by services external to the providing organisation, consider making the endpoints be redirected through a Gateway.

Response Codes: consider sending and being prepared to process HTML response codes.

#### Disadvantages

Few systems are developed with integration APIs, even less with webhook endpoints.

There is no industry standard or common pattern for the development of either webhooks or their payloads, so each can have their own quirks and associated development and testing costs.

Without acknowledgement messages, the server may believe that clients have been notified.

### Integration APIs

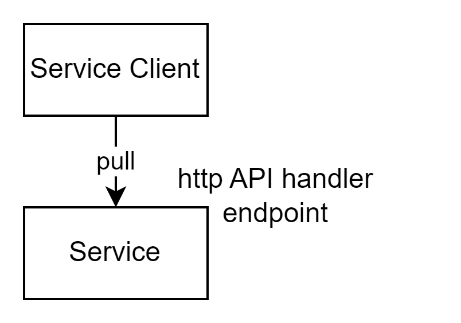


Figure 2: API integration approach

Application Programming Interfaces (APIs) serve as an interoperability-based bridge between systems, allowing one system to request information from or make changes to another.

API protocols typically follow one of two primary approaches: invoking Procedures or managing Resources.

Remote Procedure Call (RPC) based APIs enable systems to execute a wide range of commands with varying signatures. In contrast, Resource-based APIs focus on enabling Create, Read, Update, and Delete (CRUD) operations on defined system resources.

Generally, RPC-based protocols like SOAP offer greater complexity and flexibility, making them valuable for interactions within the same environment. On the other hand, the straightforward nature of Representational State Transfer (REST) has made it especially useful as the standard for APIs exposed to external third-party service consumers.

REST constrains remote clients to use one of four basic HTTP verbs (operations) on a specified resource. For example, a client retrieves information by sending a GET request to a resource, or submits or modifies data by using POST, PUT, or DELETE verbs as necessary.

Given the scope of this document in addressing third-party integrations, and the move away from use of SOAP for current development, the focus will remain solely on REST APIs.

Note:  
The approach follows a [PULL](#Term_PULL) as opposed to a [PUSH](#Term_PUSH) pattern, providing more specific control to the recipient as opposed to the provider.

#### Advantages

Unlike Webhooks, services that offer REST endpoints do not rely on the need to access the remote client system directly.

Because REST APIs are based on HTTP/S, requests can be routed through a Web Application Firewall (WAF), providing an additional layer of security.

The widespread adoption of REST-based API architectures means that a wealth of market resources, tools, development patterns, and standards exist to simplify development and testing.

Beyond OData, there is no universally accepted approach for implementing qualifiers to verb-based actions; common functionalities such as filtering, projecting, sorting, and paging are handled according to individual choices.

#### Considerations

Channel Security: APIs must operate over HTTP or HTTPS to ensure secure, reliable communication between distributed clients—even across public internet infrastructure. Adhering to these standards promotes interoperability and protects data integrity throughout transmission.

Authentication and authorisation: Implementing robust authentication and authorization solutions—such as OAuth or JWT—is vital for verifying client identities and managing secure access to API resources. Authentication may be required at both the gateway and the service.

Gateways: long standing public APIs should be available through redirection provided by an API gateway. They enable rerouting of API traffic, thereby supporting maintenance and evolution of services without requiring clients to modify their integrations, a crucial outcome when APIs are available outside of the organisation. However, in enterprise settings, where systems are rarely replaced, their materialised benefits may remain more hypothetical than materialised. They also introduce latency, infrastructure, bandwidth consumption and licensing costs.

Rate limiting: To prevent abuse and ensure fair distribution of resources, APIs should enforce limits on the frequency of client requests.

Lifespan Management: APIs should clearly communicate their intended support lifecycle, signalling deprecation and end-of-life milestones well in advance. This transparency enables consumers to plan migrations and adaptations effectively.

Versioning: Since consumers may continue to use older API versions despite repeated prompts to upgrade, establishing clear versioning practices is essential. Defining the API version as a header or argument—rather than embedding it in the path—ensures consistent access to resources across all versions.

Queryability: Adopting standardized frameworks like OData, developed by leading technology organizations and managed by OASIS, provides a strong foundation for designing and consuming RESTful APIs, supporting robust and flexible query capabilities.

Idempotency

: Ensuring idempotency is a critical aspect of robust API design. It is recommended to use standards-based approaches to achieve idempotency, rather than relying on custom or ad hoc solutions. When RESTful principles are applied, as in the case of OData, change operations are inherently idempotent and do not interfere with the effectiveness of caching mechanisms. Conversely, in non-REST frameworks such as GraphQL, all operations are transmitted via POST requests, which precludes standard HTTP caching. As a result, these frameworks do not natively support idempotency or cacheability, requiring more compute and storage infrastructure and costs, emphasizing the importance of adhering to established standards whenever possible.

Principled design: Once third-party services begin using an API, making subsequent changes becomes challenging. To avoid widespread disruption, it is essential to keep data schemas concise and well-segregated, ensuring that changes in one domain do not cascade into others. Entities that endure over time, such as “people,” should remain distinct from relationships (like “studentAt”) and profiles (“studentProfile”), rather than being combined into a single, mutable entity (e.g., “Student”).

Monitoring: Comprehensive monitoring and logging practices enable thorough tracking of API usage, enhance troubleshooting, and support system reliability.

Error Handling: Consistent error handling, accompanied by clear documentation and standardized error messages, streamlines the troubleshooting process for external developers and supports successful integrations.

#### Disadvantages

The inherent simplicity of RESTful APIs presents certain design challenges, particularly when operations on resources must themselves be treated as distinct resources—so-called “operation requests.” This requirement often leads to the creation of workarounds in API architecture.

When implementing qualifiers for verb-based actions, using established standards is generally preferable to adopting unique or novel solutions. OData, formalized as ISO/IEC 20802-1:2016, serves as a robust standard for this purpose. However, its adoption has faced competition from non-standard, open-source protocols such as Facebook’s GraphQL.

Service designers aiming to increase third-party engagement are frequently compelled to offer APIs compatible with both OData and GraphQL. This dual approach reflects the evolving landscape of RESTful API design, where compatibility and adherence to best practices remain central to fostering broader market adoption.

### UI APIs

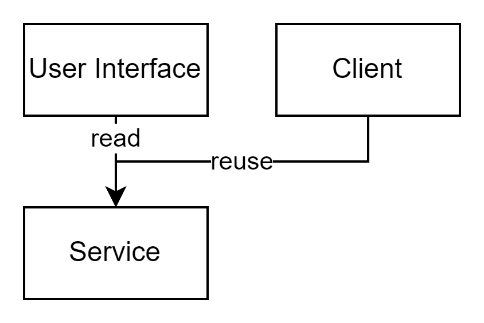


Figure 3: UI API Reuse Approach

APIs created primarily for user interface purposes are commonly referred to as UI APIs, and should not be confused with integration APIs. These APIs may exist within a system without being specifically designed to facilitate integration with external services. For instance, many UI APIs are intended for use with Asynchronous JavaScript and XML (AJAX) requests, which allow data to be fetched and updated without reloading the entire web page—an important feature for Single Page Applications (SPAs).

Unlike integration APIs, UI APIs are typically restricted to a single tenancy and a specific context, such as an individual school, and operate directly within the currently active user session.

#### Advantages

Separating the user interface from the system logic enhances overall maintainability and reduces both server resource usage and input/output demands. The user interface, being the most visible and quickly outdated component of a service, often needs frequent updates to keep pace with evolving user expectations and technological advancements. By maintaining the user interface as a distinct codebase, teams can refresh or upgrade it efficiently without requiring any modifications to the underlying business logic.

The API is not intended for use by other organisations or departments – so APIs can be updated without introducing breaking changes that can’t be fixed.

UI APIs are specifically designed for use within the user interface of a system, not for external integration across organizations or departments. By maintaining this focused operational boundary, updates to the user interface and its supporting APIs can be made in tandem, allowing the system to remain responsive to changing user needs and evolving requirements. This separation supports both agility and reliability within the intended context of use.

#### Considerations

Data Security and Privacy: ensure that API endpoints and integration methods comply with data protection regulations and best practices. Carefully manage authentication, authorization, and the storage or transmission of sensitive information, especially when dealing with multiple user contexts or direct datastore access.

Context Management: when APIs are restricted to single-tenancy or context, plan for effective management of user sessions and context switching. Consider provisions for maintaining accurate lists of context identifiers and processes for updating or synchronizing them securely.

Efficiency and Resource Utilization: evaluate the efficiency of solutions, particularly when repeated sign-ins and context cycling are required. Minimize redundant queries and optimize the refresh mechanisms to avoid unnecessary server or network load.

Scalability: anticipate growth in user base, contexts, and integration needs. Design APIs and integration approaches that can scale gracefully without introducing bottlenecks or excessive administrative overhead.

Reliability and Maintainability: separate UI and business logic to enable independent updates and reduce system brittleness. Ensure that the integration approach—whether API-based, UI-driven, or agent-driven—can be maintained over time as requirements and technologies evolve.

Security Risks of Workarounds: assess the security implications of using multiple user accounts, context cycling, or direct datastore access. Each workaround may introduce new vulnerabilities or attack surfaces that must be proactively managed.

Monitoring and Auditing: implement robust monitoring and logging, especially when using service agents or non-standard integration methods. Maintain clear audit trails to support incident response and compliance.

Standardization and Compatibility: favour established standards (like OData) to improve interoperability and reduce technical debt. Balance support for popular but non-standard protocols (such as GraphQL) according to business needs and market demands.

User Experience: consider the implications of your integration or API approach on end users, particularly regarding latency, reliability, and support for modern application patterns like SPAs.

It is essential to maintain a clear distinction between UI APIs and integration APIs. These interfaces are designed to serve different stakeholders, and their requirements often diverge significantly. Attempting to merge or conflate these APIs may create the illusion of efficiency, but any perceived short-term savings are likely to be misleading. Ultimately, maintaining separate, purpose-built interfaces ensures both stakeholder needs and long-term system flexibility are properly addressed.

#### Disadvantages

In situations where no Interoperability API is provided, and service clients wish to use these UI APIs for integration, they must repeatedly sign in and out to access the required contexts. This approach is inefficient and time-consuming, particularly when only a single request per context is necessary.

This approach assumes that one user account is a member of all required contexts. To manage this, the service client must be provisioned in advance with a complete list of context identifiers, ensuring that a single user account has access to each. Alternatively, the client may need both a list of context identifiers and unique credentials for multiple accounts—one for each context. Both scenarios present distinct security risks.

Maintaining up-to-date lists of context identifiers and user accounts requires a dedicated refresh mechanism to keep pace with changes. However, even with regular updates, the client must iterate through the full list, regardless of whether any changes have actually occurred on the server. Most queries, therefore, will return no new data, resulting in inefficient use of resources.

In summary, this method is not only brittle but also significantly increases the demands on both time and system resources.

## Non API Based Integrations

When integration-specific APIs are not available – and trying to use a workaround by using UI-focused APIs proves too challenging and unreliable -- it becomes necessary to explore less optimal, non-API solutions.

### Physical Transfers

An option is to manually transfer data from source system to target system via a portable datastore. Such as a thumb drive.

#### Advantages

There are no significant advantages to using thumb drives for manual data transfer in this context.

#### Considerations

When direct access to the source data is not possible—such as when the data resides within infrastructure controlled by a third party that does not permit the installation of ETL agents—manual transfer may become the only feasible solution, despite its significant drawbacks. While ETL agents are generally preferred for data integration tasks, the lack of access permissions can necessitate resorting to physical, manual methods.

Security: the drive should be encrypted.

Data Integrity: Manual processes are prone to human error, such as incorrect file selection or accidental deletion, which can compromise the accuracy and completeness of transferred data.

Version Control: There is an increased risk of version mismatches, as multiple manual copies may be created, making it difficult to ensure the target system receives the most current data set.

Compliance: Transporting data via physical media may violate organisational policies or regulatory requirements regarding data handling, storage, or cross-border movement.

Scalability: Manual transfer methods do not scale well with growing data volumes or frequency of transfers, quickly becoming unsustainable for larger operations.

Resource Allocation: Staff time and attention are diverted from higher-value activities to supervise and execute manual transfers, impacting overall productivity.

#### Disadvantages

Confidentiality: Manual data extraction requires direct access to the production system, which exposes sensitive information during the process of creating and transferring CSV files.

Auditability: There is no reliable way to track or audit who accessed the system or data, leading to potential gaps in accountability.

Security: Portable media such as thumb drives are at risk of being lost or misplaced during transit, jeopardizing the security of the transferred data.

Data Structure: The export process typically flattens the data structure. This can undermine relational data integrity or require a specific processing sequence to correctly import the data into the target system.

Timeliness: The time lag between data extraction and confirmation of successful receipt and usability can be significant, affecting the overall reliability of the transfer.

Effort: Manual processes demand substantial preparation, such as arranging for appropriate access rights. This makes manual transfer the least desirable option due to the high effort required.

Repeatability: Since the process is entirely manual, it is slow and labor-intensive. Participants are less inclined to repeat these steps, which negatively impacts the freshness and relevance of the data.

### SFTP

An alternative to manual transfer methods is the use of a Secure File Transfer Protocol (SFTP) server, which allows secure data exchanges between systems belonging to different organisations.

#### Advantages

This method offers modest improvements over physical transfers by reducing reliance on portable media and enabling direct electronic transfer. SFTP can facilitate quicker data movement and enables basic access control through user credentials.

#### Considerations

Although SFTP provides a faster means of transfer, many of the challenges associated with manual processes persist. The process still requires both parties to set up and maintain SFTP services, with appropriate credentials issued for data deposit and retrieval. Coordination is needed to ensure that the correct files are uploaded and downloaded, and to manage user permissions for secure data access.

#### Disadvantages

The disadvantages are largely the same as those found with physical transfer methods. SFTP does not address issues of auditability, repeatability, or the need for manual involvement. Data structure integrity may still be compromised if files are exported in flat formats, and delays or errors can occur if files are transferred incorrectly. Furthermore, while SFTP is more secure than using physical media, it remains dependent on the proper management of credentials and secure channels, exposing the process to risks if not handled diligently.

### Direct ETL

An option is to using an integration service to Extract Transform and Load (ETL) data from

### Integration Agents

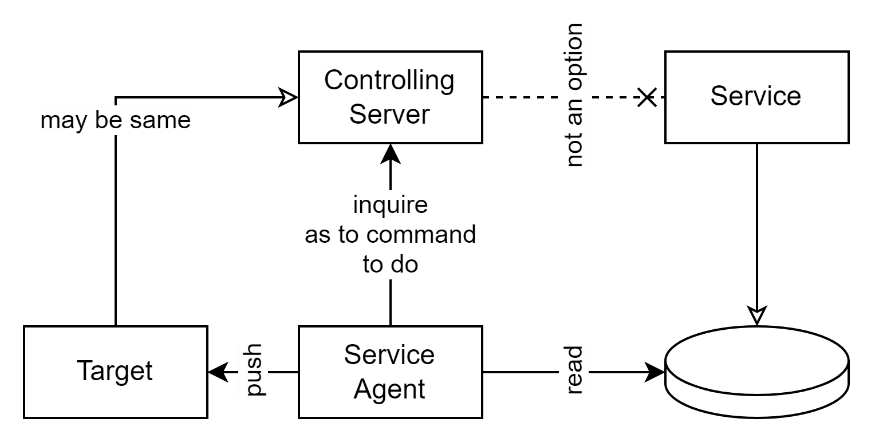


Figure 4: ETL Service Agent Approach

An option is to bypass conventional interfaces and access internal components directly with a service agent.

A common example involves datastores—most often relational databases—which can be queried if a service agent is installed on a device within the same network as the target system.

Service agents may function in two primary ways: they can perform tasks independently, following instructions that were set during deployment; or they can operate under real-time remote control, receiving commands from a central controlling server.

This approach enables direct interaction with system internals, providing an alternative pathway for data access or process automation when standard integration methods are not feasible.

Note:  
A remotely controlled service agent effectively changes the process from a [PUSH](#Term_PUSH) to a [PULL](#Term_PULL) approach, similar to what an integration API approach provides.

In both cases, communication between server and target destination, and/or communication between service agent and controlling server, if any, are required to be over secure channels.

Additionally, the service agent must identify itself using credentials. These credentials are required to be initially provided to the client in a secure manner, then persisted by it in a secure manner (e.g.: using an installed or default system encrypted keystore)[[3]](#footnote-4).

#### Advantages

This method introduces additional complexity in both implementation and ongoing management but can be effective when direct integration APIs are unavailable.

Azure provides ready made solutions for these types of tasks: both Azure Data Factory and Data Fabric components.

#### Considerations

Agreement: if the source system is in infrastructure owned by another organization, agreement is required first to install a service runner. This is not a given.

Setup: the remote organisation should create a user specific to the consuming service, associate it to permissions on custom Views.

Operational Complexity: Deploying, updating, and maintaining service agents across diverse environments can increase operational overhead, especially when scaling to many systems.

User Access and Authorization: Strict controls must be in place to ensure only authorized personnel can deploy or operate agents, reducing the risk of misuse or unauthorized data access.

Security: Direct database access introduces significant security and compliance challenges. Agents must be configured to use the principle of least privilege, and all credentials should be managed with robust encryption and secure storage practices.

Custom Development Effort: Since there is no standard convention for agent implementation, each use case may require custom design, development, and extensive testing.

Network Requirements: Reliable and secure network connectivity between agents, controlling servers, and target databases is critical. Firewall rules, VPNs, and secure tunnels may be needed to facilitate communication.

Change Management: Modifications to the underlying databases or network architecture may impact agent functionality. Processes must be in place to manage compatibility and version control.

Error Handling and Recovery: Agents should be designed to gracefully handle errors, provide meaningful alerts, and recover from common failures to avoid data loss or corruption.

Actions performed by service agents may not be logged in a permanent manner, providing only limited auditability as only a byproduct of diagnostic logging. Establishing an audit trail may be difficult to meet regulatory compliance.

The target tables may not have sufficient attributes to be usable for watermarking (LastModified). If it doesn’t a full import of data is required to determine new records. The attribute may be in local time requiring conversion to global/universal date/time.

The target table may physically delete records, requiring a full sync to compare against previously imported data to stop deletions.

Performance Impacts: Intensive queries or poorly optimized agents can affect the performance of production systems. Careful scheduling and resource allocation are recommended to minimize disruption.

Compliance: Ensure that data extraction and transmission adhere to relevant legal, regulatory, and contractual requirements, particularly when handling sensitive or personal information.

#### Disadvantage

Before deploying service agents for ETL operations, installation of a suitable runner or device in the target environment is essential. The operating system must be compatible with the execution of the required agent (for example, the Data Factory Self Hosted Integration Runner requires Windows). Arrangements such as a Statement of Work (SoW) or other agreements with the administrators of the remote server may be needed.

A dedicated database user must be created within the RDBMS for agent operations. This user should either be linked to a mutual Identity Provider (IdP) or, if not possible, the user's sensitive credentials need to be securely communicated and persistently stored by the consumer.

Custom Views and/or Stored Procedures tailored to the ETL tasks are required, and these must be associated with the designated database user for proper access and execution.

In the absence of an established convention or pattern, each ETL task requires unique design, development, and comprehensive testing. For instance, a typical workflow might include executing one or more queries against the target database, exporting results in a delineated format such as CSV, then packaging and transmitting these files to a specified destination.

In certain implementations, service agents can be remotely managed from a central service. This service provides instructions regarding execution timing and the specific data to retrieve, enhancing flexibility and oversight of ETL operations.

### Integration Agents through Tunnel Agents

A diagram of a cloud service

AI-generated content may be incorrect.

Figure 5: Integration via a Tunnel Agent

In situations where installing a traditional integration agent is not permitted—such as when the remote organisation only supports opensource systems, preventing the use of tools like Azure Data Factory Self-Hosted Integration Runtime (SHIR) which only is installable on Windows OS managed devices —an alternative must be considered. A viable solution is the implementation and use of a virtual tunnel agent.

If permission can be secured, a virtual tunnel agent (for example, Cloudflare’s “cloudflared” agent, which is compatible with both Windows and Linux environments) is installed on a device within the remote organisation’s network, in proximity to the source database. This agent initiates an outbound connection to its cloud-based controller (such as Cloudflare), establishing a secure virtual tunnel.

In the consumer’s network, an integration agent is installed. Database queries are then routed through Cloudflare and the remote tunnel agent, ultimately reaching the target data source.

#### Advantages

Cross-Platform Compatibility: Tunnel agents often support a wide range of operating systems, including Linux, macOS, and Windows, making them adaptable to diverse IT environments.

Minimal Firewall Reconfiguration: Because tunnel agents typically initiate outbound connections, there is often no need to open inbound firewall ports, reducing security risk and simplifying setup.

Rapid Deployment: The installation and configuration process for tunnel agents is generally straightforward and can be completed quickly, enabling faster project rollouts.

Granular Access Controls: Many tunnel agent solutions offer detailed access control features, allowing organizations to restrict agent operations to specific resources or times.

Reduced Need for VPNs: Secure cloud based tunnels can eliminate the necessity for complex VPN infrastructure, streamlining network architecture and reducing maintenance overhead.

#### Considerations

Trust: The remote organisation must have a high level of trust, as they are required to install an agent within their network.

Security: Security is primarily managed by the cloud provider, such as Cloudflare.

Isolation: The source organisation may need to set up a virtual network to isolate the source data.

Setup: the remote organisation should create a user specific to the consuming service, associate it to permissions on custom Views.

Monitoring: It is likely that monitoring will be implemented to ensure the agent’s access is restricted to authorised systems only.

Maintenance: The tunnel agent will require periodic updates and ongoing maintenance.

Dependency on Third Parties: Reliance on external services (like Cloudflare) introduces potential risks, such as service outages or changes to terms that may impact operations.

Performance and Latency: Data transmission through virtual tunnels can introduce additional latency or bandwidth constraints, potentially affecting ETL job performance.

Troubleshooting Complexity: Diagnosing connectivity or data flow issues can be more complex due to the involvement of multiple networks and external services.

Compliance and Auditing: Additional due diligence may be required to satisfy internal and external audit or regulatory requirements for data handling via virtual tunnels.

Change Management: Any updates to cloud agents, endpoints, or network configurations may require careful coordination between teams across organisations.

#### Disadvantages

Increased Complexity: Setting up and maintaining virtual tunnels involves multiple components and can complicate system architecture.

Security Risks: While security measures exist, opening tunnels can expand the attack surface and require diligent oversight.

Performance Overheads: Data routed through tunnels can experience reduced speeds and increased latency, affecting job efficiency.

Reliance on External Services: Dependence on third-party providers means that any changes in their offerings or outages can disrupt operations.

Troubleshooting Difficulties: Diagnosing and resolving connectivity or data flow problems can be more challenging due to the layered infrastructure.

Compliance Challenges: Ensuring that data handling via tunnels meets internal and regulatory standards may require additional audits and processes.

Coordination Requirements: Updates or changes often require careful collaboration between multiple teams and stakeholders, potentially slowing response times.

# Conclusion

Integration APIs are the preferred method for system communication, however constraints may sometimes necessitate alternative approaches.

In such scenarios, direct database access for Extract, Transform, Load (ETL) operations serves as a practical substitute. When deciding between scheduled ETL tasks and remotely managed service agents, the use of a centralised agent, which can be directed to run specific operations and retrieve targeted data, is generally favoured.

Appendices

Appendix A - Document Information

### Authors & Contributors

Author: Sky Sigal, Solution Architect

### Versions

* 1. Initial draft.
  2. Glossary diagrams added.

1.0 Updated for Te Rito

1.1 Updated for Pourato

### Images

[Figure 1: Web hooks based integration 6](#_Toc207181951)

[Figure 2: API integration approach 8](#_Toc207181952)

[Figure 3: UI API Reuse Approach 11](#_Toc207181953)

[Figure 4: ETL Service Agent Approach 15](#_Toc207181954)

[Figure 5: Integration via a Tunnel Agent 18](#_Toc207181955)

### Tables

**No table of figures entries found.**

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Sandy Britain, Enterprise Architect |  |
| Russell Campbell, Project Manager |  |
| Amy Orr, Data Domain Architect |  |

### Audience

The document is technical in nature, but parts are expected to be read and/or validated by a non-technical audience.

### Structure

Where possible, the document structure is guided by either ISO-\* standards or best practice.

### Diagrams

Diagrams are developed for a wide audience. Unless specifically for a technical audience, where the use of industry standard diagram types (ArchiMate, UML, C4), is appropriate, diagrams are developed as simple “box & line” monochrome diagrams.

### Terms

Refer to the project’s Glossary.

**Asynchronous JavaScript and XML (AJAX)**

: a front-end development technology. A key technology for developing [SPA](#Term_SPA) based service clients.

**Application Programming Interface (API)**

: a system’s interface for remote access by another person. Preferably it authenticates authorises, audits, validates access.

**Create Retrieve Update Delete (CRUD)**

: an acronym for the primary operations on data stores.

**Character Separated Values (CSV)**

: a text-based information exchange format.

**Extract, Transform, Load (ETL)**

: an acronym for a common process of transferring information between systems, where one or both parties do not have an integration API.

**Graphical User Interface (GUI)**

: a form of a system’s [User Interface (UI)](#Term_UI). Traditionally [WIMP](#Term_WIMP).

**GraphQL**

: a often used but non-standards based open source approach to qualifying some [REST](#Term_REST) operations. See [OData](#Term_OData).

**IT**

: acronym for Information, using Technology to automate and facilitate its management. IT is a subset of ICT.

**ICT**

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities.

**JavaScript Object Notation (JSON)**

: a serialisation format for describing objects used for communicating between systems.

**Long polling[[4]](#footnote-5)**

: Long polling is used in real-time web applications to achieve near-instantaneous communication between clients and the web services. It is particularly useful in chat and messaging applications where real-time updates are crucial. When used by service agents behind firewalls it has the advantage that it is permitted to call out to the external command server, whereas the command server would be blocked calling into an agent.

**Minimum Viable Product (MVP)**

: A deliverable that meets the minimum set of functional and non-functional requirements expected by stakeholders.

**Missing Valuable Planning (MVP)**

: the state of Agile projects to which no project planning has been applied.

**ODATA**

: an international standards-based method of qualifying [REST](#Term_REST) operations (e.g., filtering, sorting, projecting, paging). Contrast with [GraphQL](#Term_GraphML).

**OpenAPI**

: a standard for describing [REST](#Term_REST) based interfaces in a machine-readable format.

**PULL**

: an integration approach where the service client requests what it wishes for from a service.

**PUSH**

: an integration approach where the service pushed to service clients without confirmation that the server desires it.

**Remote Procedure Call (RPC)**

: an API architecture pattern for [API](#Term_API)s that permits the invocation of Procedures/Functions. Contrast with [REST](#Term_REST), an [API](#Term_API) architectural pattern that provides methods to manipulate Resources instead.

**REsource State Transfer (REST)**

: a messaging protocol used to communicate between systems.

**Simple Object Access Protocol (SOAP)**

: a messaging protocol used to communicate between systems. See [REST](#Term_REST).

**Singe Page App (SPA)**

: a modern approach to web solutions where the whole site, minus data is transmitted to the browser, calling back for data as needed to be displayed. SPAs are noted for improving responsiveness while decreasing server resource utilisation as well as IO bandwidth, therefore cost.

**Textual User Interface (TUI)**

: a console form of [UI](#Term_UI).

**User Interface (UI)**

: an interface for use by system users (as opposed to a remote system). May be a [TUI](#Term_TUI) or [GUI](#Term_GUI).

**Web Application Firewall (WAF)**

: a security firewall specialised with analysing web-based HTTP/S traffic.

**Windows, Icons, Mouse, Pointer (WIMP)**

: an acronym for a type of [GUI](#Term_GUI).

1. To avoid design brittleness and maintenance requirements it is recommended that links avoid passing information. [↑](#footnote-ref-2)
2. Sometimes colloquially referred to as ‘reverse APIs’. [↑](#footnote-ref-3)
3. Use of encrypted cleartext is not appropriate for sensitive data. [↑](#footnote-ref-4)
4. [What is Long Polling? | PubNub](https://www.pubnub.com/guides/long-polling/) [↑](#footnote-ref-5)