Decision Point for API and Service Implementation Architecture

Published 18 July 2022 - ID G00772596 - 53 min read

By Analyst(s): Gary Olliffe

Initiatives: Application Architecture and Integration for Technical Professionals

Delivering APIs is core to flexible, distributed applications and integration, but there are diverse architectural choices. Application technical professionals responsible for designing APIs and services should use this Decision Point to select the best architecture for each service.

Overview

Key Findings

- Modern application delivery is inherently service-oriented. Modern applications are composed of diverse services with multiple granularities, and they are exposed via public or private mediated APIs.
- Teams delivering APIs and services will use a mixture of implementation architectures and service granularities and must balance complexity, capability and cost of delivery.
- APIs, including REST and other styles, must abstract their consumers from the underlying implementation architectures, whether it be integration with existing applications and data or newly developed software.
- Although microservices architecture builds on long-established service-oriented architecture (SOA) principles, it is not always the best service implementation architecture for your APIs and services.

Recommendations

As a technical professional responsible for application architecture and integration to deliver APIs and services, you should:

- Design services using an API contract-first approach to ensure they meet consumers needs before you determine the implementation architecture.

- Decouple and abstract API consumers from service implementation details by differentiating your interface (API) from your service implementation using the mediated API pattern.
- Choose the service implementation architecture on a service-by-service basis to avoid a one-size-fits-all approach that compromises your software delivery approach, platform capabilities and ongoing operations.
- Decide on the right service implementation architecture by assessing the service function, complexity, change cadence and platform availability using this Decision Point.

Decision Point Question

What architecture should you use to implement new APIs and services?

Modern application software architecture requires the design and delivery of new APIs. These APIs make your data and functionality available to enable application composition, customer-facing experiences, system and partner integration requirements as well as many other use cases. Each API is an interface definition that needs an implementation. Use this Decision Point to choose the optimal architecture for the services that implement and support these APIs. This means you should use the Decision Point multiple times to identify the right architecture for different APIs and services within a given project or product delivery. Because services should be independent and autonomous, each service requires its own decision on architecture and granularity, based on its context.

Decision Overview

APIs and the services that implement them are critical to modern application delivery. Application services decompose and encapsulate the business logic and data persistence of your applications, providing an abstraction between source systems and consumers. They allow you to provide application capabilities and make data accessible through cross-platform interfaces that may include REST, GraphQL, gRPC, Webhooksor Apache Kafka. These interfaces can then be used with software composition, automation and development tools to meet business needs. For more information see Choosing an API Format: REST Using OpenAPI Specification, GraphQL, gRPC or AsyncAPI.

However, you should not build APIs and services for the sake of being service-oriented. In all cases, APIs and services should fulfill a business need that delivers value. Your service implementations can support a variety of scenarios simultaneously, including:

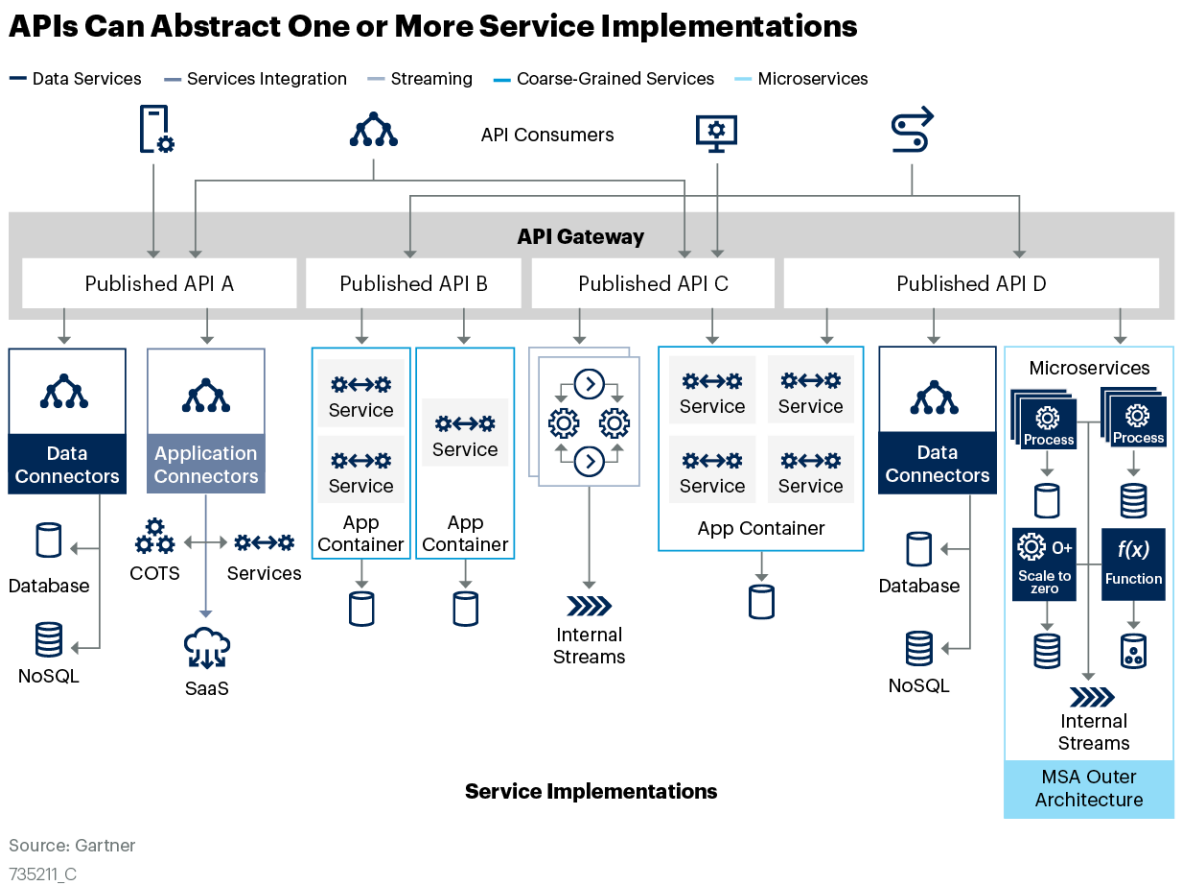
- Publishing external APIs (either public or private) to support B2B integration
- Publishing internal APIs to support system integration, composition and data distribution
- Delivering multiexperience customer and employee apps, including mobile, web, chat and voice
- Enabling tactical application delivery using low-code and no-code development platforms
- Application modernization to support business agility and flexibility
- Application migration to cloud platforms
- Process automation and optimization through business process management (BPM) and robotic process automation (RPA) integration

These examples all have a common goal: to enable creation of business applications, processes and solutions. Composition requires access to application services and data sources via an API — the interface that allows services to be invoked, consumed or accessed programmatically. Whether your APIs are “web APIs” using HTTP or use other protocols and technologies, they should be designed and managed with the API consumer as the focus (see How to Design Great APIs).

APIs should be managed as products or platforms with technical and operational contracts upon which developers and their own applications, support and operations can rely. The API should encapsulate the services that implement it, to separate the concerns of the service consumer and the service provider. As viewed by the API consumer, a single API represents a logical service that is implemented by one or more service implementations, as shown in Figure 1.

Download All Graphics in This Material

Figure 1. APIs Can Abstract One or More Service Implementations



In all cases, we recommend that the APIs exposing these services to consumers should be mediated and governed with appropriate policy using an API gateway or API management solution (see Decision Point for Mediating API and Microservices Communication, How to Evaluate API Management Solutions and How to Delivery Sustainable APIs).

When you adopt an API-centric approach you must recognize that you are building service-oriented architecture (SOA), albeit one not constrained by historical SOA standards and technologies. However, when implementing services to deliver APIs, the following principles of SOA (taken from A Guidance Framework for Applying SOA Design Principles, first published by Gartner in 2009) still apply and are important to follow:

- **Service orientation** — Encapsulating discrete system capabilities into technical assets that multiple consumers can use
- **Separation of concerns** — Separating different aspects of a software system so they may change at different rates

- **Loose coupling** — Reducing dependencies between system components so that changes to one don't adversely impact others

These principles should guide the design and architecture of your services. This Decision Point focuses on selecting appropriate architecture for creating these application service implementations.

Architectural Context

The only effective approach for designing successful APIs that expose services is to take a consumer-centric interface-first approach, and to enforce loose coupling between the API design and the service implementations. If you take this approach, you have the flexibility to build the services behind your APIs using a variety of architectures, choosing the architecture best-suited for each type of service. You also have the flexibility to change the service implementation independently of the API definition. This separation of published API from service implementation is best implemented using the "mediated API" architecture pattern. This allows not only separation of API from service, but also separation of runtime access policy from application logic.

Choose the architecture for building each service carefully. The wrong service implementation architecture can lead to overengineered platforms that don't fit the time and resources available.

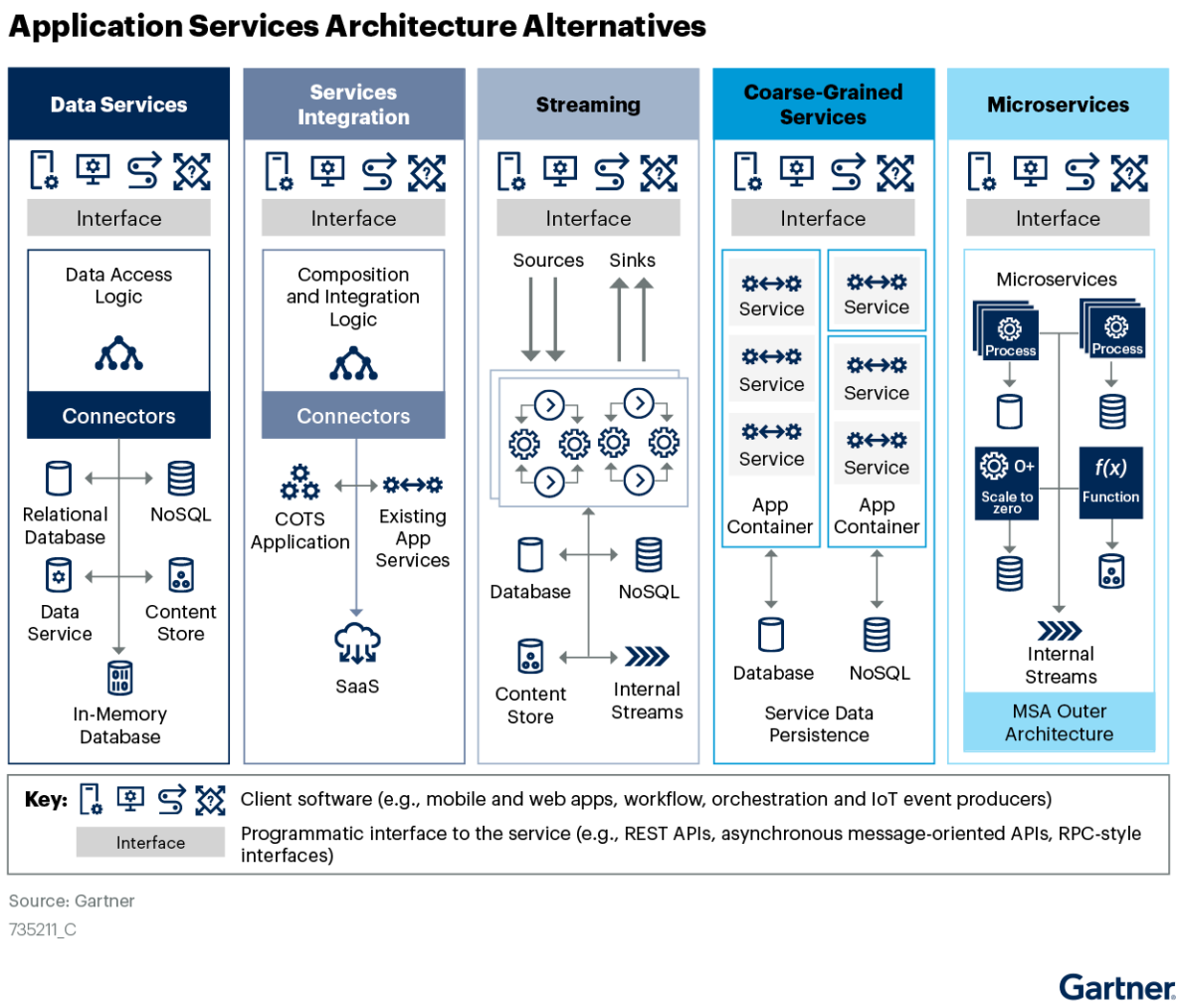
This Decision Point helps you choose the right architecture for implementing your application services from these common alternatives:

- **Data services architecture** — Using data services frameworks or data virtualization tools to compose and expose existing data products (i.e., data at rest in databases, caches, data warehouses, data lakes or other repositories) through service interfaces.
- **Services integration architecture** — Using an integration platform that provides a declarative model for composing, connecting and integrating existing services and data to create composite services. This includes service enabling and encapsulating existing applications as a step toward modernization.

- **Streaming architecture** — Using streaming platforms and reactive programming frameworks that use a stream-oriented model to ingest or expose event data from one or more sources and distribute it to one or more destinations (sinks).
- **Coarse-grained services architecture** — Developing new services implementing coarse-grained capabilities, bundling multiple services into a single unit of delivery or building services into an existing application to encapsulate and expose functionality. These capabilities are commonly exposed via REST using mature application frameworks and server platforms that encourage bundling of multiple service implementations into coarse-grained deployments.
- **Microservices architecture (MSA)** — Developing new services that decompose and isolate application functionality into tightly scoped, independently developed and deployed microservices (see How to Succeed With Microservices Architecture Using DevOps Practices for further details).

The diversity of business requirements driving new application services means that you should anticipate implementing services using more than one of these architectures. Use this Decision Point each time you have a new class or cluster of similar application services to implement. Figure 2 shows the alternative architectures.

Figure 2. Application Services Architecture Alternatives



Source: Gartner
735211_C

Gartner

Decision Tool

When selecting a service's implementation architecture, you must balance a variety of functional and nonfunctional criteria to make an optimal decision. The evaluation criteria for this Decision Point exclude functional requirements and constraints related to deploying new application infrastructure. For example, the criteria cannot take into account the cost, procurement time and deployment effort for any new application platform technology you may require, as these vary widely.

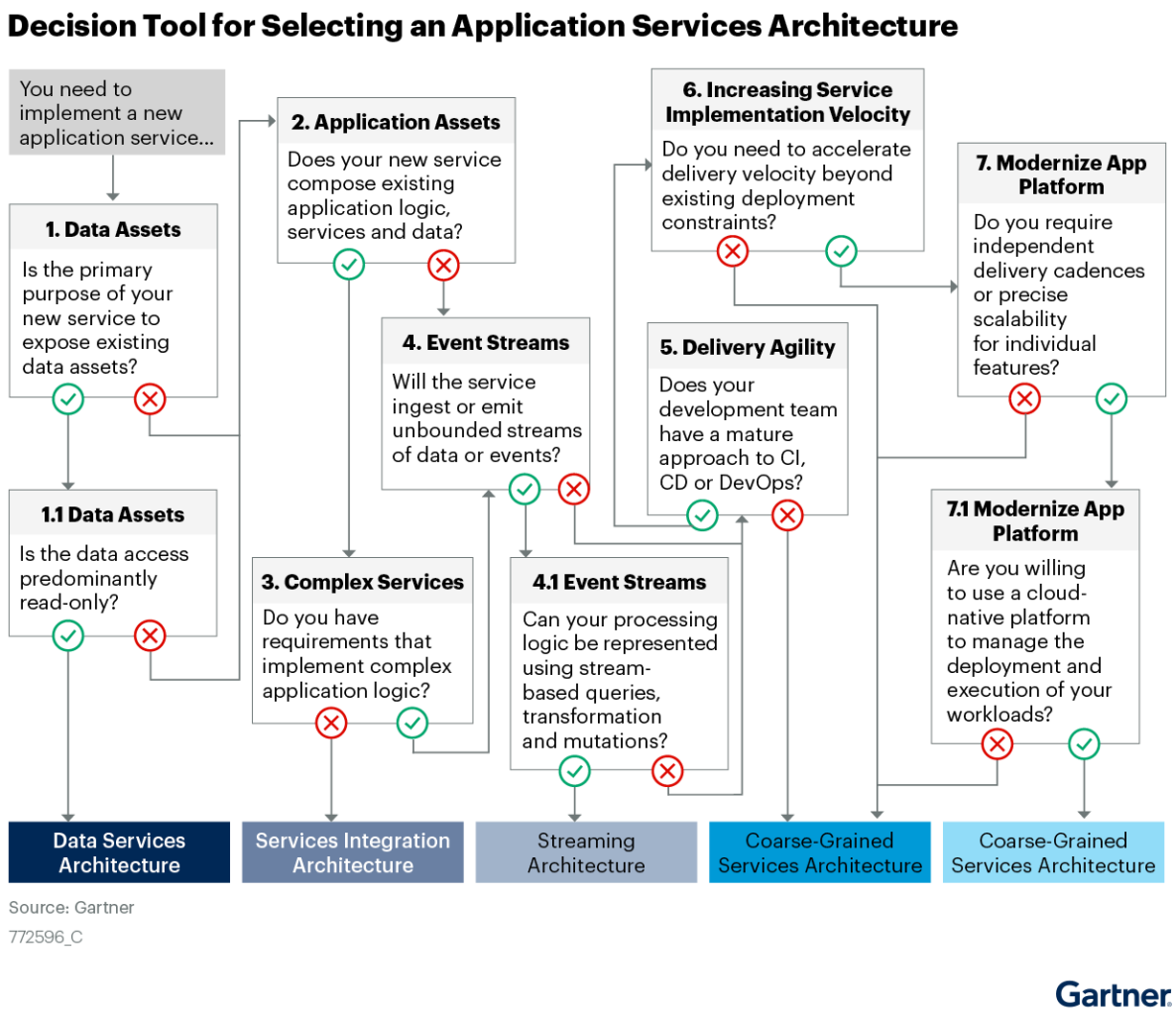
Teams with these types of constraints should evaluate whether there will be repeated opportunities to benefit from new platform capabilities and build the business case for implementing them outside of short-term project demand. More details on the platform decisions can be found in Workload Placement in Multicloud IaaS+PaaS Environments and Solution Criteria for Public Cloud Kubernetes Services.

You should use the following capability criteria to select the most appropriate implementation architecture for your application services. Will your service:

1. Expose and/or combine existing data assets
2. Access and/or compose existing application assets
3. Implement complex services
4. Process unbounded streams of data and events
5. Support delivery agility (agile, continuous integration [CI], continuous delivery [CD] and DevOps)
6. Enable service implementation velocity
7. Require modernized application infrastructure

To help architects choose the appropriate architecture for a given set of nonfunctional requirements, the decision-making process is summarized in the decision flow shown in Figure 3.

Figure 3. Decision Tool for Selecting an Application Services Architecture



Starting in the top left-hand corner of Figure 3, use this decision flow to filter your options for the set of application services requirements you are evaluating. The decision flow does not take account of the weightings for your requirements and constraints, but aims to provide a clear direction based on the critical criteria that affect the suitability of each architecture. The decision flow is also organized to make decisions based on leveraging existing assets earlier in the flow. These flows lead to data services architecture and services integration architecture decisions.

Note that this decision does not denote a choice of product type, since there are some products that support multiple patterns. For example, some data virtualization solutions offer limited service integration capabilities, and some application integration solutions offer the ability to connect to data sources and expose them as services. Integration frameworks can also be used to deliver integration services without the need to use an enterprise integration platform. See Choosing Application Integration Platform Technology for more details.

Principles, Requirements and Constraints

Requirements and Constraints

Selecting the right architecture for your services will have a significant impact on the success of your services during all phases of their life cycles. The following sections describe each of the criteria used in this Decision Point, and are followed by an evaluation of each architecture's support for the criteria.

Accessing Existing Data Assets

Decision Tool Steps 1 and 1.1

Every organization has data, and the value of that data may be locked in your data warehouses, data marts, databases or content stores and accessible only via data extracts, business analytics tools or via the applications that own the data. The data may also be fragmented across many repositories and technology platforms.

In many cases, these data assets are held in systems of record or systems of differentiation (see Use Pace Layers to Align Your Application Strategy With Your Business Strategy). Packaging and serving this data in a flexible yet manageable way is common when supporting use cases that include:

- Providing access to master data consistently, and on demand, to business applications and processes
- Enabling innovation projects to access existing data assets in a controlled way, but with minimal governance overhead (e.g., supporting self-service developer or integrator access to approved data assets)
- Enabling the use of data in low-code or no-code application and integration platforms
- Minimizing data duplication and replication between application domains
- Exposing data to business partners, customers and other stakeholders to support transparency, integration and automation of external business processes

Existing data may be inaccessible due to compliance or security reasons, but it may also be locked away because the mindset of the IT organization is based on historical application models of data ownership and stewardship.

In the era of composite applications, the stewardship and ownership of the data your composite applications use is spread across a number of different stakeholders. Using the appropriate master data management practices can help identify value in these data assets, support data discoverability and help control access (see Implementing the Technical Architecture for Master Data Management).

You can find further information about exposing, composing and integrating data assets in Assessing the Relevance of Data Virtualization in Modern Data Architectures and Solution Criteria for Data Integration.

Accessing Existing Application Assets

Decision Tool Step 2

Every organization has a portfolio of commercial off-the-shelf (COTS) SaaS and custom-built applications. These applications often represent a huge sunk cost and an ongoing operational commitment for the organization. Although many organizations are modernizing and rationalizing their application portfolios, there will always remain a need to provide services that access existing application assets. The three most common scenarios for integration services are:

1. Service enablement of legacy applications to allow their functions and data to be exposed over more modern API patterns.

2. Mediation of existing service interfaces to support a consumer-centric API design — for example mapping simple object access protocol (SOAP) services to REST resources.

3. Composing multiple services using orchestration logic to implement a new business service. The services being composed may expose up-to-date APIs, legacy interfaces or a combination thereof.

For older applications this requires integration technology, such as a platform enterprise service bus (ESB) or integration framework, that is capable of integrating with the underlying technology of the application or its platform (often using traditional software development kits [SDKs]). Modern applications are likely to offer APIs defined by the vendor or the development team. However, these interfaces often do not meet the needs of the service consumers your organization has identified. They require some form of API mediation. This mediation may include protocol mediation, payload transformation/translation, service aggregation and service orchestration.

Common use cases for exposing existing application assets in the form of new application services include:

- Enabling near-real-time integration between business applications as part of composite application business processes.
- Enabling innovation projects to access existing application assets in a controlled way to support digital business requirements, such as mobile integration and web API implementation.
- Decoupling the life cycles of faster-changing service consumers from slower underlying service providers.
- Creating process integration with business partners, customers and other stakeholders to support transparency and automation of business transactions.
- Creating “shareable services” from application-specific service interfaces that were designed for intra-application integration.
- Creating an easier-to-use catalog of APIs with consistent protocol and payload attributes by mediating interfaces to existing applications that are disparate and diverse.

The following documents provide further information on integrating with existing application assets:

- Choosing Application Integration Platform Technology
- Choosing Data-, Event- and Application-Centric Patterns for Integration and Composition

Implementing Complex Services

Decision Tool Step 3

Application services implementations can range from simple data access, like exposing data via an API, to services that implement highly complex business logic. This diversity means that not all implementation architectures are suited to creating a full range of services, and identifying complex processing requirements will help to differentiate the alternatives.

At the time of selecting an implementation architecture for your application services, assessing the complexity of service can be a qualitative judgment. Complex processing requirements might include compute-intensive calculations based on application data from across the domain model, or heavily conditional and hierarchical algorithmic logic leading to complex business component or object models.

The key purpose of this decision step is to eliminate integration-centric architectures from scenarios that have complex functional requirements. Integration tools excel at connectivity, process orchestration and composition, but they are not appropriate for implementing complex business logic.

Processing Unbounded Streams of Data and Events

Decision Tool Steps 4 and 4.1

Modern businesses are data driven, and you can treat changes to that data as a continuous stream of new information. Examples include sensor data from connected devices, clickstream data from customer-facing websites and event notifications from business processes, such as order processing or customer support.

This information needs to be processed or analyzed before it can be acted upon. Streaming architecture provides a set of capabilities to handle the processing of high-volume, high-value streams. Your service is processing unbounded streams when:

- The interface to your services is designed to ingest discrete events or messages from its clients (e.g., your client is a gateway service aggregating sensor data from a fleet of connected devices).
- You emit discrete events intended for a specific client (e.g., a stream of events notifying clients of changes to an airline schedule).
- The position or order of events in the stream (relative to other events) is an important context for processing.

The interface provides a way for data to be collected into or consumed from the stream. The interface may be a RESTful API but it could also be exposed using other protocols and technologies, including Apache Kafka, gRPC (client streaming), GraphQL subscriptions or WebSockets. The interface itself should also be explicitly described using AsyncAPI, OpenAPI Specification (OAS) 3.1 (Webhooks), GraphQL schema or a similar format. The interface should be defined as a set of topics and payload schemas supported by an event broker to which client (data sources) connect. The implementation of your service will subscribe to one or more streams of events to process them. The service may publish new streams of derived or complex events as a result of its processing.

The following documents provide further details on streaming architecture patterns:

- Choosing Event Brokers: The Foundation of Your Event-Driven Architecture
- Streaming Analytics in the Cloud: A Comparative Analysis of Amazon, Microsoft and Google
- Essential Patterns for Event-Driven and Streaming Architectures

Supporting Delivery Agility

Decision Tool Step 5

Agile development practices are now widespread and well-understood (see Solution Path for Agile Transformation). Achieving the full benefits of agile development is highly dependent on eliminating or reducing delays in taking code from development to production. This has led to the emergence of continuous integration (CI), continuous delivery (CD) and DevOps practices that help to reduce delays and streamline the delivery life cycle without compromising on software quality.

This criterion assesses how well a given architecture supports these practices for delivering new application service implementations. An optimal environment for these practices will support deployment automation, isolated deployment structures to reduce risk of side effects, reliable and efficient portability of application assets between environments, and support strategies such as canary testing and blue-green deployments. The following documents provide further information on adopting agile development and related practices:

- Essential Skills for Agile Development

- Keys to DevOps Success
- 12 Ways Your Agile Adoption Will Fail

Increasing Service Implementation Velocity

Decision Tool Step 6

Every development effort benefits from work moving faster through the system. Lowering the effort required to create and deploy new services enables developers to improve their delivery cadence. However, development team throughput may need to be traded off against other factors, such as cost, technical capabilities and resource availability. An architecture that allows you to deploy new application services with a lower level of effort — and hence faster delivery cadence — can help your project succeed.

This criterion assumes that you do not have any established deployments of the architecture alternatives. It takes into account the complexity and effort required to deploy and configure the architecture platform and implement your new application services.

When you have one or more of these architectures (and the platforms to support it) established, you should rank those alternatives more highly for this criterion. For example, if you have an existing, well-structured platform for building, deploying and managing microservices, then the time and effort required to bring a new service to production may be relatively low, increasing implementation productivity. See Solution Path for Applying Microservices Architecture Principles for more details.

Modernizing Application Platform

Decision Tool Steps 7 and 7.1

The architecture for new application services is often constrained by the application platform you currently have available. If you have access to a recently modernized application platform that is appropriate for your service, you should take advantage of it. However, your choice of service architecture and application platform should be aligned.

Traditional enterprise application platforms are still valid choices for teams with existing licensing investments, skilled people and application requirements that leverage the features of these platforms. Such platforms include Java Platform, Enterprise Edition (Java EE and, more recently, Jakarta EE) application servers and Microsoft Internet Information Services (IIS) and the ASP.NET runtime. [1] However, there is a continuing trend toward alternate approaches, specifically for:

- Services that do not benefit from the more arcane features of these traditional platforms (so those unused features become a burden in terms of licensing/maintenance costs, platform complexity and execution efficiency).
- Services that can be deployed to application platforms in the cloud or on-premises that provide alternate solutions to the nonfunctional requirements (such as scalability and resilience) fulfilled by traditional application servers.
- Services that have specific functional or nonfunctional requirements that can be implemented more efficiently or with significantly higher performance, using more-specialized platforms that support capabilities such as:
  - Dynamic provisioning
  - Functional programming
  - Actor/reactor patterns
  - In-memory data processing
  - Nonblocking input/output (I/O)

These alternatives include adoption of technologies such as:

- Lightweight framework stacks (e.g., Spring Boot [Java], Quarkus [Java], GoMicro, Micronaut [Java], Moleculer [JS]) and runtimes (e.g., .NET, Go, Node.js)
- Application platform as a service (aPaaS) providers or frameworks (e.g., Microsoft Azure App Service, Salesforce Heroku, VMware Tanzu Application Service, Red Hat OpenShift)
- Container management platforms and related tools (e.g., Azure Kubernetes Service [AKS], Amazon Elastic Container Service [Amazon ECS], Red Hat OpenShift Kubernetes Engine, VMware Tanzu Kubernetes Grid)
- Serverless or function PaaS (e.g., AWS Lambda, Azure Functions, Azure Container Apps, Google Cloud Run, and Cloudflare Workers)
- Contemporary messaging and event-streaming technology (e.g., Apache Kafka, Apache Pulsar, NATS, RabbitMQ) and cloud event brokers (e.g., Amazon Kinesis, Azure Event Hubs, Azure Service Bus and Google Cloud Pub/Sub)

- In-memory data grids and data stores (e.g., Apache Geode, GridGain, Hazelcast, memcached, Oracle Coherence, Redis)

The adoption of such technologies can be associated with the adoption of microservices architecture (MSA). However, the benefits are not limited to organizations adopting MSAs. Achieving the benefits of MSA requires the adoption of agile methods, continuous delivery and DevOps, as well as investment in a robust set of platform capabilities that support the increased complexity of a highly distributed and dynamic operating environment. Teams that do not (yet) require the extreme flexibility, scalability and agility of MSA can still benefit from some of these contemporary application platforms and technologies by using them for more coarse-grained service applications, where appropriate.

Application infrastructure modernization will also impact the technology choices for the services that it supports. For those services, you have to ensure the languages, application runtimes and platforms are compatible and that you have the appropriate platform operations capabilities in place (see Using Platform Ops to Scale and Accelerate DevOps Adoption).

The following documents provide further information on modernizing your application architecture and infrastructure:

- Solution Path for Applying Microservices Architecture Principles
- How to Succeed With Microservices Architecture Using DevOps Practices
- Designing Services and Microservices to Maximize Agility
- Assessing Red Hat OpenShift Container Platform for Cloud-Native Application Delivery on Kubernetes
- Solution Criteria for Public Cloud Kubernetes Services
- Decision Point for Selecting Virtualized Compute: VMs, Containers or Serverless
- How to Empower Technical Teams Through Self-Service Public Cloud IaaS and PaaS

Alternatives

The alternative architectures covered by this Decision Point are grouped as follows:

- Data services architecture
- Services integration architecture
- Streaming architecture
- Coarse-grained services architecture
- Microservices architecture

Selecting the right implementation architecture for your application services is a critical step toward a successful implementation, but the variety of requirements and the options available mean that there is not a single solution for all situations. There are trade-offs to be made in choosing each alternative.

Data Services Architecture

Exposing data assets using services remains a common and valuable requirement. Data services are sometimes frowned upon as being more data- and content-oriented than service-oriented, but they remain a valuable architecture option that can help extract significant value from existing data held in systems of record or systems of differentiation. Data services may also be part of a data fabric implementation supporting self-service access to data resources across an organization.

The implementation technology for a data services architecture includes three approaches:

1. **Data services frameworks,** such as the various REST and GraphQL frameworks including Apollo, [2] Prisma, [3] Spring Data REST, [4] OData Web API, [5] Apache Olingo [6] and Agrest. [7] These frameworks provide development teams with the ability to bind service implementations to data stores and expose data as plain old XML (POX) or JavaScript Object Notation (JSON).

2. **Data virtualization and federation platforms,** such as Denodo Platform, [8] Oracle Data Service Integrator, [9] Red Hat JBoss Data Virtualization [10] and TIBCO Data Virtualization. [1] [1] These support aggregation of data from multiple sources into a virtual data model that can then be exposed as a set of services. Note that some data virtualization products also support services integration architecture.

3. **Data API platforms** provide services and API generation from existing data stores, such as Appery API Express, [12] CData API Server, [13] DreamFactory, Hasura, [14] SlashDB, [15] IBM API Connect and Broadcom CA Live API Creator.

The following sections highlight the strengths and weaknesses for the data services architecture approach to implementing new services based on the Decision Point criteria.

Strengths

- **Accessing existing data assets:** The key capability of this architecture is to aggregate and/or transform data assets and expose them via one or more service interfaces. Note that in some cases it will be necessary or desirable to implement a data integration hub architecture to aggregate data resources into an intermediate database for sharing and serving via API.
- **Increasing services implementation velocity:** Where your requirements can be met by data services, this architecture has a low-build effort when compared to other alternatives.

Weaknesses

- **Accessing existing application assets:** This architecture can leverage some application logic if embedded in database triggers and stored procedures.
- **Implementing complex services:** Not suited to complex service logic because data services should be focused on create, read, update, delete (CRUD) operations with persistent data resources.
- **Processing unbounded streams of data and events:** Data services frameworks and tools are focused on the serving of data held in existing repositories, such as relational and document databases. They cannot ingest or process data streams. Some technologies used for data services also have the ability to subscribe to event brokers (e.g., CA Live API Creator).
- **Modernizing application infrastructure:** Decouples databases from consumption and aggregates data supporting database and data model modernization activity, but does not directly support modernization of application infrastructure.
- **Supporting delivery agility:** Can support agile iteration of service features, but data virtualization options use a traditional “platform management” approach (i.e., a shared platform to support multiple services).

When to Use a Data Services Architecture

Use a data services architecture when you need to expose data sources and data products via API to streamline integration and reduce data duplication. Implementing new services using a data services architecture enables an organization to extract value from existing data assets by making them more consumable and available for composition into new applications and processes. Although this approach does not support implementation of complex new service logic, it can support the creation of data services that decouple applications and processes that consume the services from the underlying representation of business and application data. When exposed through a managed API, this approach can enable controlled self-service access to valuable corporate data in a way that does not unduly inhibit the rapid delivery requirements of opportunistic and adaptive projects.

Where data virtualization technology is used to expose these services, it is possible to provide a more consistent and aggregate view of data from a variety of repositories and applications. This model can help improve the productivity of new service implementations by allowing data management or application teams, familiar with the data, to manage the optimization of data services. This will reduce the time-to-value for new services.

Where data service frameworks are used, application development and maintenance efforts must be accounted for (compared to the configuration effort when using data virtualization). This may be justifiable for scenarios where there are a limited number of data services to be implemented, or where the services will remain consistent over an extended period of time. GraphQL is emerging as a popular API standard for exposing data-centric services, particularly where the API's consumers are modern multiexperience apps and where the data-centric services must span multiple data stores.

This architecture is best-suited to synchronous, read-only access to existing data assets.

Solutions that implement CRUD access are possible but, in most cases, modifying application data requires execution of existing business logic. In these cases, you should first consider a services integration architecture. If your requirements are predominantly read-only (either by number of service interfaces or by planned transaction volumes), consider splitting the read and write behavior into separate services using the command query responsibility segregation (CQRS) design pattern. [1] [6] Then, use a data services architecture for the "query" (read) service.

Services Integration Architecture

Integration middleware can be used to implement new services, typically based on existing application and data assets. A variety of product categories can support the services integration architecture. The key requirement is that they can connect to existing resources and compose or mediate access to expose new application service interfaces that can be consumed by downstream applications. These product categories include:

- **Application integration platforms** (including enterprise service buses [ESBs]) deployed onto self-managed infrastructure on-premises or using IaaS.
- **Integration platform as a service (iPaaS)** provided entirely in the cloud or using a hybrid model (i.e., integration execution takes place in your self-managed infrastructure).
- **Integration frameworks** (such as Apache Camel and Spring Integration) that can be used by application developers to implement common integration patterns: GraphQL frameworks such as Apollo, GraphQL Ruby and GraphQL-dotnet are also being used to compose and integrate multiple back-end services into a single GraphQL API. However, these tools do not explicitly provide support for orchestration of services.
- **Data virtualization platforms** that support connectivity to, and composition of, messaging and service interfaces exposed by applications.

In each case, the technology provides a declarative model for composing, connecting and integrating existing services and data via multiple protocol-level and/or application-level connectors and adapters. Also note that there is some overlap between these technologies and those used for data services architecture. A single product or framework can be used to implement multiple architecture patterns.

Services integration architecture is also one way to service-enable existing applications, especially when the existing application cannot be modified to implement and expose "native" service endpoints. This approach may also require the integration technology to connect to the existing application through a proprietary or legacy interface (for example, using a customer connector that uses the applications' SDK).

For further details on the technology choices for services integration, see Choosing Application Integration Platform Technology.

The following sections highlight the strengths and weaknesses for the services integration architecture approach to implementing new services based on the Decision Point criteria.

Strengths

- **Accessing existing data assets:** Services integration tools have database and data service protocol connectors. However, if you need to maintain quality and consistency across a large dataset, the data services architecture is more appropriate.
- **Accessing existing application assets:** Integration tools are available with a wide variety of protocol- and application-specific connectors and adapters.

Weaknesses

- **Implementing complex services:** Services integration architecture can support complex compositions, mediations and orchestrations. However, complex logic will drive capacity-related costs and may cause platform lock-in. Avoid turning your integration platform into a container for application logic.
- **Processing unbounded streams of data and events:** Application integration tools can be connected to event brokers and stream tools. The implementation will typically handle discrete messages rather than provide stream-oriented processing.

When to Use a Services Integration Architecture

Use a services integration architecture when you are primarily connecting to and combining functionality and data managed in existing applications to meet new requirements. The ability to reuse existing application assets as part of your service-oriented architecture or API platform is a critical capability. Integrating existing services and service-enabling existing applications to deliver new service capabilities is a required capability for any project that crosses application or organizational boundaries.

Services integration architecture enables an organization to derive further value from existing investments through reuse. It can also help to automate and optimize processes and decision making across application and organizational domains. This model also allows those existing assets to be combined with capabilities from third-party services, thus enriching and enhancing them to meet new requirements or to create more effective solutions. This approach can also be valuable for implementing or integrating message-based or event-based interactions between application services and for integrating synchronous and asynchronous domains.

The technology solutions for creating services using the services integration architecture are broad, including application integration platforms, enterprise service buses, message brokers, API management solutions and iPaaS platforms. However, the common theme is that these classes of products are mediators. They are good at enabling and managing access to services, but they are not well-suited to implementing and hosting net new application service implementations. They will help you to define and implement the logic to compose new services from existing application and data assets, but they should not become a container for your application code. Application integration tools used in this way become increasingly complex to deploy, operate and scale.

Streaming Architecture

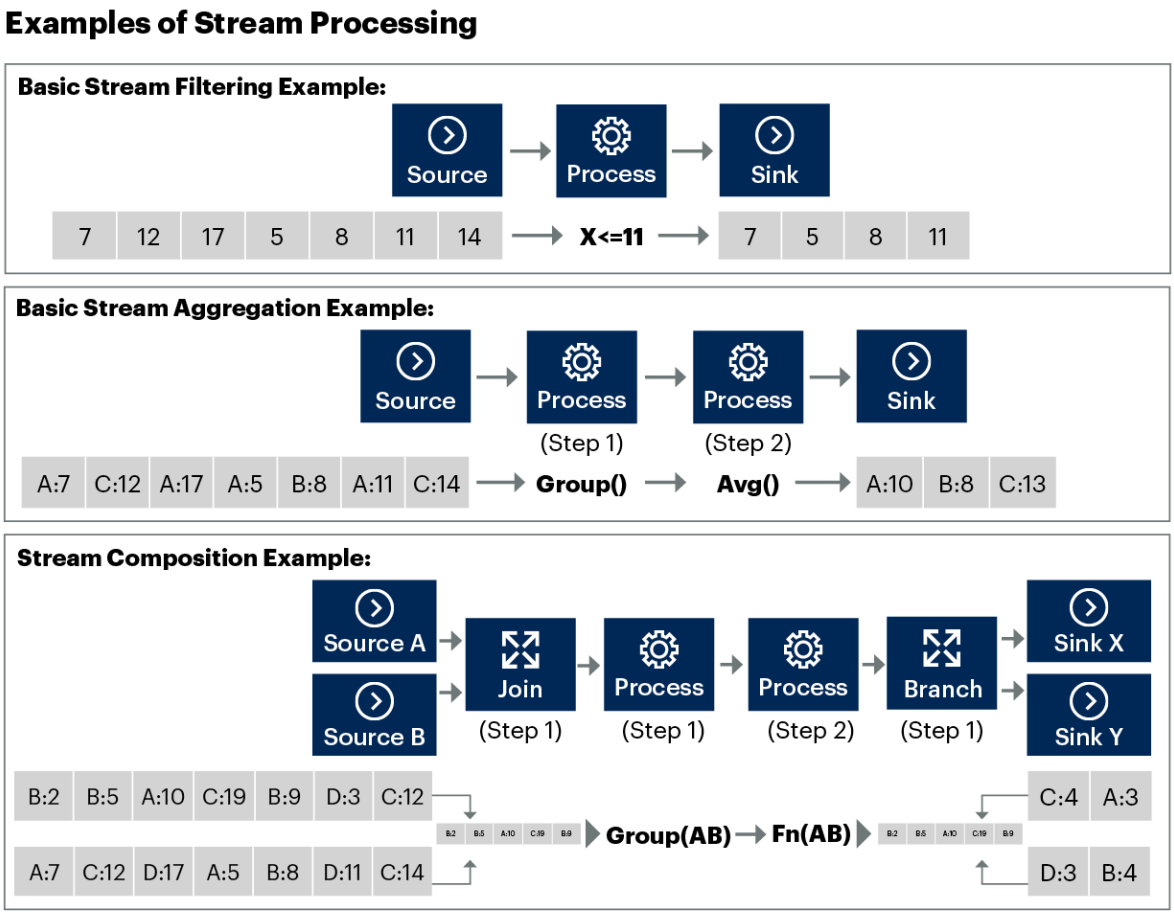
Streaming architectures are becoming increasingly relevant to application architects. This is being driven by the demands to process more data more quickly, the accessibility of streaming technology, and increasing awareness and usage of event-driven architectures.

A streaming architecture is used to build services that ingest and process streams of records or messages (which often represent event notifications) using patterns applied directly to the stream of data, rather than individual records. These services can connect to an event broker to support publish and subscribe (pub-sub) communication patterns. In this case, the API is defined by the topic structure and message schemas used to deliver messages.

Alternatively, services with streaming architectures can be exposed via API endpoints through which a stream of messages can be delivered (for example, a WebSocket interface or a Webhook interface). Services implemented using streaming architecture can also emit streams of messages either directly to an event broker or through APIs using protocols such as WebSockets, HTTP streaming and server-sent events.

Streams of events are emitted by one or more sources and processed in one or more steps that result in some output to one or more sinks, as shown in Figure 4. A streaming architecture implements its processing logic as a set of composable steps that can filter (map), transform, aggregate and join data in one or more streams. Streaming platforms and frameworks provide programming structures and runtime management that simplify the implementation of these patterns, improving productivity, quality of delivery and scalability.

Figure 4. Examples of Stream Processing



Source: Gartner
735211_C

Gartner

As with all the alternatives described in this research, it is key that the service in question has a well-defined interface. For the streaming architecture, this interface is dictated by the sources and sinks of the data streams it will process. The interface will dictate the structure of the events and how they are to be delivered. For example, when the source is an event broker, the definitions and locations of the topics through which event data will be delivered are critical components of the interface. An API definition format such as AsyncAPI can be used to describe these interfaces.

Streaming architectures can work with a variety of sources and sinks for data, including:

- **Databases and data services, whether relational, document- or graph-oriented.** When used as a source, some form of change data capture is often used to detect and emit events (e.g., IBM InfoSphere Change Data Capture [InfoSphere CDC], Oracle GoldenGate or Debezium).
- **Event broker middleware,** such as Apache Kafka or RabbitMQ
- **Internet of Things (IoT) platforms,** such as Amazon Web Services (AWS) IoT, Azure IoT Hub and IBM Watson IoT Platform
- **File-based and blob-based storage,** including network-attached storage; distributed file systems like Ceph, Hadoop Distributed File System (HDFS) or Gluster; and cloud services such as Amazon Simple Storage Service (Amazon S3) and Azure Blob Storage
- **Cloud services,** including SaaS platforms, customer engagement platforms and data brokers that expose streaming APIs

The streaming architecture approach to processing data and events can be applied in two fundamentally different ways that share high-level stream-oriented paradigms but require different supporting technology:

- **Localized stream processing** — An individual service instance uses streaming patterns to retrieve data from a source, process it and emit some result. Frameworks such as Akka Streams, Java 9 Flow API, ReactiveX, Project Reactor and Vert.x Reactive Streams provide programming frameworks for developers to apply streaming architecture to their individual service implementations. However, they do not inherently support the distributed processing of individual streams (i.e., there is no shared-state or built-in coordination between instances of a service).

- **Distributed stream processing** — A streaming platform provides coordination, orchestration and state management to enable stream processing to be scaled across multiple processes or nodes. Examples of frameworks and platforms that support this model of processing include Apache Flink (Ververica, Amazon Kinesis Data Analytics), Apache Spark (Azure Databricks and HDInsight), Apache Storm (Azure HDInsight, Cloudera) and Confluent ksqlDB. These platforms allow you to develop using the platform API and submit jobs to a cluster of dedicated nodes, running the platform, to be deployed and executed. Apache Kafka Streams also supports distributed stream processing but does not require a dedicated cluster of compute nodes. Instead, developers can build in Java and deploy their Kafka streaming services on any platform, such as in a PaaS or general-purpose container management platform like Kubernetes. These services then connect to a Kafka cluster to coordinate their activity.

The API or service-centric use cases for streaming architecture include both inbound and outbound streams, for example:

- Publishing an API that delivers event notifications that originate in internal streams/sources and have filtering and transformations applied.
- Ingesting and processing events from mobile or web apps, IoT devices or logged events in other application processes.

The following sections highlight the strengths and weaknesses for the streaming architecture approach to implementing new services based on the Decision Point criteria.

Strengths

- **Processing unbounded streams of data and events:** The reason this architecture exists is to deliver scalable, performant processing models for unbounded streams from a variety of sources.
- **Supporting delivery agility:** Stream processing platforms can be integrated into traditional CI/CD pipelines (e.g., using Jenkins), and deployments of specific stream processes can be isolated to ensure independent deployment.
- **Increasing services implementation velocity:** Decoupled components/processor and “chaining” of processing steps provide some decoupling. Streaming platforms support independent deployment of streaming jobs/processes.

Weaknesses

- **Accessing existing application assets:** This approach is not intended for application integration or composition but can be used to process event streams sourced from or synced to existing applications.

When to Use a Streaming Architecture

Use a streaming architecture when your service will be ingesting and processing or emitting continuous streams of data or events. This is distinct from message-oriented or event-driven scenarios where each discrete message or event can be treated in isolation. Streaming architecture can still be used to process events in isolation (e.g., transforming or filtering them). However, the value is made more apparent when you have requirements for grouping related events, partitioning streams, joining streams and applying windowing functions to the events.

In addition to the logical structure of the data arriving at or leaving from your service, the nature of the interface to your service is also an important consideration. Streaming platforms and frameworks support multiple sources and sinks, including sourcing events from files of various formats, database tables, message brokers and event brokers. But when building application services and APIs, you must decide how you will define, publish and host your interface.

If your goal is to ingest a constant stream of event data from your clients, you might have an HTTP façade supporting individual event publications (e.g., POST to/devices/{id}/events) or expose WebSockets or MQTT endpoints to reduce the protocol overheads per event.

If your service will emit streams of data to clients that subscribe, you will need an event broker to manage the connections to the subscribers and deliver messages to them. Multiprotocol brokers such as Apache ActiveMQ, IBM MQ and RabbitMQ can support open standards such as Advanced Message Queuing Protocol (AMQP) or MQTT as well as proprietary REST APIs. Apache Kafka requires clients to use a supported client code library but can be fronted by a Kafka REST proxy.

Support from API management and gateway vendors for publishing and managing event-driven APIs is limited at present, so you must also consider how you will publish and govern access to these endpoints. Products such as IBM Cloud Pak for Integration (Event Endpoint Management), Gravitee, WSO2 API Manager and Axway Amplify Streams have support for event streaming APIs defined using AsyncAPI.

Coarse-Grained Services Architecture

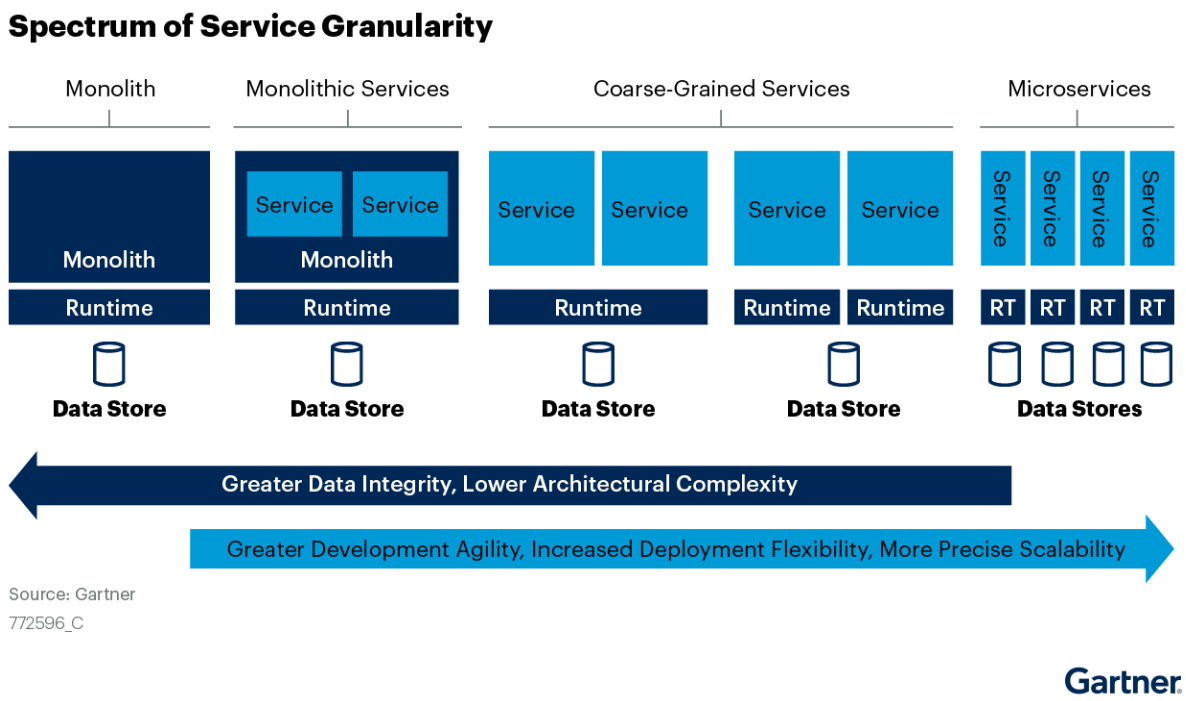
Building service-oriented applications (whether SOAP- or REST-based) using mature application server platforms has been the preferred approach to developing new application services for more than a decade. In this model, application services are typically built and deployed using monolithic project structures that bundle service implementations into a single deployment package. This approach has high levels of dependency on shared packages (Java), .NET assemblies or other component frameworks at development time.

The platforms supporting this model include Java application servers (both Java EE and traditional Spring applications), the Microsoft .NET platform (Internet Information Services [IIS], Windows Communication Foundation [WCF] and ASP.NET), Ruby on Rails and many others. This alternative includes services implemented using either SOAP or RESTful interfaces because we are comparing implementation architecture rather than interface style and structure.

If (or rather when) demand for change to the software accelerates, these architectures can become a constraint and drag on the delivery life cycle. Releases require the build, test and deployment of a larger and more complex codebase, increasing risk and effort for each update, so feature updates and enhancements are batched up to reduce the overhead. Well-architected monolithic services with strong CI/CD and test automation can still achieve high delivery cadence, but they retain operational dependencies and lack deployment flexibility. Do not discard a simpler, more monolithic architecture unless you can manage the additional complexity and reap its benefits.

Service granularity is best viewed as a spectrum, as shown in Figure 5. The coarse-grained services architecture alternative recognizes that the principles of loose coupling, high cohesion and independent deployment espoused by MSA can be applied in a more pragmatic way to reduce but not eliminate these release challenges.

Figure 5. Spectrum of Service Granularity



This is not a new architectural approach or paradigm, simply a clarification that service orientation means that we can decompose the deployment of application functionality to address the challenges of modern application delivery. Coarse-grained services architecture allows for packaging of one or more services into cohesive units of deployment.

Solutions built using this architecture have coarse-grained service and data domains. Services in a given domain commonly share a data persistence architecture with a common data model (i.e., a suite of application services all connect the same application database that hosts a single data model with strong referential integrity). In addition to this, a single deployment package (the unit of deployment) will commonly implement a suite of service interfaces for a specific, cohesive application domain.

The following sections highlight the strengths and weaknesses for the coarse-grained services architecture approach to implementing new services based on the Decision Point criteria.

Strengths

- **Implementing complex services:** Well-suited to complex service logic implemented in code (e.g., Java, .NET, Ruby, C++) or supported by code libraries and components linked at compile or runtime.
- **Supporting delivery agility:** Can support agile iteration, deployment and test automation, but uses a traditional "platform management" methodology. Traditional application platform licensing can inhibit CI/CD/DevOps adoption because all running instances of the software used through the pipeline may need a license. For instance, deploying duplicate instances for blue-green deployment patterns may require additional licenses, even though the duplication is transient.

Weaknesses

- **Processing unbounded streams of data and events:** Traditional services frameworks have limited support for stream-oriented processing, but there is some support for streaming interfaces such as HTTP response streaming and server-sent events.

When to Use a Coarse-Grained Services Architecture

Use a coarse-grained service architecture when your change-velocity requirements can be met through good software engineering and testing discipline. Structuring application logic into discrete services by applying service-oriented architecture (SOA) principles is not a new concept. For many teams, this will be the default approach to developing new application functionality. In some cases, these services are broad in scope and complex in implementation. They may be implemented as separate deliverables to be shared by multiple business initiatives, or they may be part of larger (composite) application implementation projects.

This architecture alternative represents the approach favored by IT teams seeking to control technical diversity and take advantage of well-understood, well-supported technology for delivering application functionality as services.

An enterprise-class application development framework, a runtime container, and a disciplined (and preferably agile) development methodology together deliver the best balance of risk, effort and reward for building application services in large teams. This is also the case in environments where standardization is favored over technology specialization. Developers and architects with the skills and experience to deliver coarse-grained service architectures are now widely available, whether working in Java, Microsoft .NET, PHP, Ruby or other common frameworks.

In addition to supporting technology and platform maturity, this approach is also best-aligned to delivering services within a typical IT organization structured around layers of the technical architecture (application layer, middleware layer, data layer, infrastructure layer, etc.). Whether you are using traditional waterfall/sequential, highly iterative or agile application development processes, building coarse-grained application services that are tested and deployed in highly coordinated releases allows you to focus your resources on delivering new functionality.

If you are approaching the limits of delivery velocity and are mature in agile development and CI/CD practices for your monolithic applications or coarse-grained service architecture deployments, evaluate whether the cultural and technical changes required to adopt MSA are worthwhile.

Microservices Architecture

Microservices architecture (MSA) has moved from being a contender a few years ago to being the most dominant services architecture pattern for new development and modernization. MSA applies SOA principles and introduces additional disciplines to the decomposition and isolation of application functionality, resulting in independently developed and deployed microservices. This decomposition enables flexibility in development life cycles and precise scalability at runtime, but brings with it the additional complexity of continuous delivery and distributed system management.

Some of the attributes of an MSA are:

- Microservices implement fine-grained bounded contexts, [17] and each owns and manages the data that it exposes and encapsulates (i.e., there is not a single physical data model shared by multiple services).

- Microservices are scoped to implement a single responsibility, to be loosely coupled from their peers and to be internally cohesive (such that most functional changes will be limited to one service); this does not mean that they have to be small in terms of code or footprint.
- The inner architecture of each service can be simple, but the architecture of the platform needed to support MSA is complex to enable automation, load-balancing, interservice communication, state management, observability and security.
- Each service can be implemented using the most appropriate development frameworks, runtimes and data persistence. This enables independent, service-specific language and database choices that can be used to optimize performance or simplify implementation.
- Each service is implemented as an independent unit of deployment, usually using a lightweight embedded web container. For example, Eclipse Jetty or Apache Tomcat for Java, ASP.NET Core, or a lightweight runtime such as Node.js or Go.
- Services communicate using open standards through synchronous and asynchronous channels, and the architecture favors event-driven choreography over process-driven orchestration patterns.

The benefits of MSA are realized when business needs for agility are met. This requires that services can be built and managed on independent, high-velocity life cycles, and application capacity requirements can be met by precisely scaling up or down the number of instances of specific microservices as needed. Your computational capacity and performance requirements should inform the style of platform you use to run your microservices, and you should consider the following:

- **VMs and IaaS** — Use these where individual services will benefit from vertical scaling and (relatively) coarse-grained capacity increments as you scale horizontally.
- **aPaaS or cloud-native application platforms** — Use these when you favor developer productivity and simplified application runtime management. Examples of aPaaS include Azure App Service, AWS Elastic Beanstalk, Google App Engine and Salesforce Heroku. Examples of cloud-native application platforms include Red Hat OpenShift and VMware Tanzu Application Platform.

- **Container platforms and services based on Kubernetes** — Use these when you need fine-grained control over container deployment and when you need to deploy other containerized software (OSS or third-party) alongside your own services. Examples include services like Amazon Elastic Kubernetes Service (Amazon EKS), Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE) as well as platforms like Rancher and VMware Tanzu Kubernetes Grid.
- **Serverless functions or containers** — Use these when you have volatile workloads that can benefit from the "zero-floor" scaling to avoid idle infrastructure capacity and rapid and dynamic scale out. Examples include AWS Lambda, Azure Functions, Azure Container Apps and Google Cloud Run.

The following sections highlight the strengths and weaknesses for the microservices architecture approach to implementing new services based on the Decision Point criteria.

Strengths

- **Modernizing application infrastructure:** Allows you to fully leverage application infrastructure (outer architecture) capable of deploying and managing many microservice instances. Use lightweight app containers, OS containers, container as a service (CaaS), function platform as a service (fPaaS) or application platform as a service (aPaaS).
- **Supporting delivery agility:** CI, CD and DevOps practices are a requirement for successful MSA implementations. MSA was defined to make CD and DevOps practices more efficient and reduce the delivery risk of discrete changes.
- **Increasing services implementation velocity:** Building a robust platform for MSA delivery (outer architecture) and the processes and organization structure to take advantage of it is a significant challenge to initial adoption. However, once these are established, building new services and delivering changes to existing services should be high-velocity and low-inertia.

Weaknesses

- **Accessing existing data assets:** Microservices demand independent and encapsulated domain-driven data models that are unlikely to be found in existing data structures. Relaxing this constraint may be the right and pragmatic approach, leading to coarse-grained services where there is increased coupling.

When to Use a Microservices Architecture

An MSA could be the right choice for your new application services if the following conditions are met:

- You have functional requirements where there is a high degree of ongoing change and a direct benefit from reducing code-to-production cycles.
- You are willing to deal with the complexities of distributed and eventually consistent business transactions and processes across microservice boundaries.
- You have nonfunctional requirements that demand precise scalability and rapid provisioning and deprovisioning of capacity.
- You are willing and able to invest in the implementation and support of a supporting platform with cloud characteristics (in particular, self-service support) while adopting new patterns for service design.
- You have identified application services functional domains that will be able to show measurable business benefit from decomposing into independently deployable component services that can be developed, deployed and scaled independently.
- You have successfully implemented test and release automation and continuous delivery for less distributed and dynamic systems.
- You are willing to organize team responsibilities around services rather than architectural layers, and have delivery teams that are accountable for the running service and not just the code following DevOps principles.
- Your development teams are well-versed in agile practices, delivering consistently and able to respond to changes from sprint to sprint.
- You have automated testing capabilities and have embraced a “shift-left” approach to testing, where developers are writing test code and using practices such as test-driven development (TDD) and behavior-driven development (BDD).
- You are using infrastructure-as-code practices, including immutable instances in production.

Choosing a microservices architecture should not be done lightly. If your requirements for application services include highly flexible, per-service delivery life cycles, and you need to be able to deploy and scale your application capacity with precision, then the benefits may outweigh the cost and complexity involved. These costs include the effort required to establish and operate the necessary platform capabilities and the changes (and disruption) to your development and operations teams as they transition to this new delivery model for application services. For further details on the benefits and impacts of adopting MSA, see Solution Path for Applying Microservices Architecture Principles.

For teams willing and able to address these challenges head-on, MSA brings with it a great deal of agility and flexibility not found in traditional application architectures. Releases become truly continuous, with each service team deploying code when it deems it ready. Releases also become lower-risk, with the ability to support canary tests, blue-green releases and regression risk isolated in simpler, loosely coupled microservices. Scalability becomes more precise and dynamic, with individual services being scaled up and down (or out and in) to meet demand as it changes over time. Technology selection can be more flexible when needed, allowing microservices to use platforms and persistence technology that is highly tuned to their requirements.

All of these benefits can be realized. However, to do so takes a strong commitment to agile methods, cloud-native application principles, service engineering discipline, DevOps principles, automated testing and organizational alignment around business domains. It also requires an investment in building or composing and operating a complex, distributed application delivery platform.

Future Developments

APIs and services are commonplace as mechanisms for delivering application capabilities as well as exposing functionality and data for integration and composition. This has been driven, in part, by the ubiquity of cloud services, mobile experiences and digitalization of business processes, which lead to business and consumer demand for connectivity and interoperability. But there is more to come. Digital businesses require ever more connectivity, composition and integration, all delivered at an even greater pace.

Future developments include:

- **Event driven APIs —** Enabling reactive, asynchronous process composition and choreography.

- **Continued evolution of containers, Kubernetes and application PaaS** — Simplifying the platforms needed to support microservices.
- **Composable business** — Businesses demanding flexible software systems that can adapt in volatile business environments.
- **Productive Automated Governance of Services** — Optimizing governance to support increased delivery cadence and adoption of distributed API-centric and cloud-native architectures.

Event-Driven APIs

REST remains the dominant API paradigm, for both internal and external APIs. However, the trend to extend these APIs with event-driven capabilities continues at pace. Although EDA is not new, it is now finally on a trajectory that will have a broad impact on how businesses, and the systems that support them, operate. Events are dual-purpose. They can be both a trigger for one or more (re)actions to take place and a record of the changing state of a system. This allows event-driven architectures to support flexible application architectures, as well as data movement, persistence and real-time analytics, while keeping both aspects loosely coupled.

The ubiquity and accessibility of event brokers is also supporting this trend. A raft of options are available from open-source or commercial software and as cloud-provider offering and multicloud managed services. Examples include Apache Kafka, NATS, RabbitMQ, Google Cloud Pub/Sub, Solace, Confluent Platform, Azure Event Hubs, Amazon Simple Notification Service (SNS), IBM MQ and TIBCO Enterprise Message Service (EMS), but the complete list is extensive. For a more detailed analysis of event-broker capabilities, see Choosing Event Brokers: The Foundation of Your Event-Driven Architecture.

Despite these options, there are still challenges and complexities to adopting EDA, not least the lack of standardization around specific protocols and payload representations. And developers and architects comfortable with designing and building event-driven systems are still at a premium compared to those more comfortable in the "request response" patterns of REST APIs. But we see this changing through our interactions with Gartner clients. The AsyncAPI project is delivering a format for defining message and event-driven APIs in a protocol-agnostic form similar to the Open API Specification for REST. GraphQL includes an opinionated approach to event subscriptions as part of a holistic set of API interaction patterns (query, mutate, subscribe). Additionally, work is underway among the community of serverless platform vendors to establish a standard format (CloudEvents) for event data, making it easier for event publishers and subscribers to propagate events.

API providers such as Box, Dropbox, GitHub, Lufthansa Group, Mastercard, Salesforce and Sony already offer event-driven APIs using a variety of protocols including HTTP Streaming, HTTP Long Polling, Webhooks, WebSockets and MQTT. API management vendors are also gradually adding support for the governance of event-driven API.

Continued Evolution of Containers, Kubernetes and application PaaS

Application PaaS (aPaaS) continues to evolve, supporting new deployment models and containers. In parallel, container management platforms continue to become ever more PaaS-like with better abstraction and integration with development life cycle tools.

At one end of the aPaaS spectrum, vendors focused on high productivity offer low-code/no-code development capabilities and embed new composition and integration capabilities into their products to enable them to be platforms for composite application development. If your organization is using low-code or no-code tooling, you will need APIs to your services and data to simplify their integration into the apps built on these platforms. This trend toward using APIs to simplify the composition and integration of data and services into business applications in the cloud and on-premises will impact both data and application integration requirements for typical enterprise IT teams.

At the other end of the aPaaS spectrum, Kubernetes is having a huge influence. Initial Kubernetes-based offerings were focused on simplifying the deployment and operations complexity of Kubernetes. Now the needs of developers using the platform are coming into focus. Offerings like Azure Arc and Google Anthos are providing support for both operating Kubernetes infrastructure and simplifying the deployment of application platform (and other) services to that infrastructure. Red Hat's OpenShift continues to refine its developer experience, and VMware's Tanzu Application Platform provides developer focused abstractions over Kubernetes.

Although not limited to microservices architectures, these platforms provide a growing number of options that can support microservices implementations with integration of developer workflow, service mesh and telemetry functions. For further details, see Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications and Assessing Red Hat OpenShift Container Platform for Cloud-Native Application Delivery on Kubernetes

Finally, there is function PaaS (fPaaS), typified by AWS Lambda and Azure Functions, which provides stateless compute capacity that is dynamically allocated at runtime and charged based on execution time. For simple computational and for event-triggered workloads, fPaaS presents a viable alternative to the typical container-based deployment model of microservices architecture.

Composable Business

Composability is back, but it never really left. Technology innovation comes in both waves and cycles. In the early 2000s, Gartner published a variety of research on composable applications and systems. In part, these were driven by the prevalence of SOA and the emergence of improved tooling for developing and integrating software components and services. In the 20 years since, the notion of an API has evolved from something your desktop or server operating system provided to the ubiquitous pattern of access to data and services. APIs now support anything from tweets to healthcare information and bank accounts.

Technologists have been composing systems by integrating services, data and third-party API products throughout this period. However, the business impact of that composition was often buried — the composition was a means to an end, such as to reduce cost or shorten project timelines — and once completed, it changed slowly.

Now, the business environment (including commercial, nonprofit and public-sector/government) is significantly more dynamic. Even before the disruptions of the COVID-19 pandemic organizations were trying to become more dynamic and responsive to the needs of their users and customers. Now, that adaptability is an imperative for business, not just for IT. For further details see Quick Answer: What Is Composable Business Architecture?

Productive Automated Governance of Services

In the 2022 Planning Guide for Application Platforms, Architecture and Integration we describe the need for governance to become more streamlined and focused on the needs of the business and the consumer of the resources being governed. We call this productive automated governance.

The good news is that well-managed APIs are a benchmark for productive automated governance today. Using API management tools to publish APIs for developer self-service, with access and usage policies automatically enforced by the platform, enables the consumer to be more self-sufficient and adaptable. This also allows the organization and the provider to keep control of what is used and who is using it.

The less good news is that this model needs expanding into the platforms and tools used to design, build and operate the services that implement these APIs, and the data they expose. Otherwise, they cannot be as effective for event-driven APIs, GraphQL gRPC and other protocols and patterns as they are for REST APIs.

Vendors are moving in this direction, gradually adding support for these additional interaction styles and governing microservices through the service mesh they use to communicate with each other. However, until these capabilities mature and become widely available, organizations implementing increasingly complex and diverse fleets of APIs and services will need to invest in extending or building some of these controls around their existing platforms and processes. The key deliverable should be governance controls that are integrated and automated so that they allow teams delivering and using governed resources to remain productive.

Evidence

[1] Overview of .NET Framework, Microsoft.

[2] The Apollo Data Graph Platform, Apollo.

[3] Homepage, Prisma.

[4] Spring Data REST, Spring.

[5] OData Web API: A Server Library Built Upon ODataLib and WebApi, GitHub.

[6] Apache Olingo is a Java library that implements OData.

[7] Agrest, Agrest.

[8] Homepage, Denodo.

[9] Oracle Data Service Integrator, Oracle.

[10] Red Hat JBoss Data Virtualization, Red Hat.

[11] TIBCO Data Virtualization, TIBCO Software.

[12] Appery.io API Express, Appery.io

[13] CData API Server, CData

[14] Homepage, Hasura.

[15] Homepage, SlashDB.

[16] CQRS, Martin Fowler.

[17] BoundedContext, Martin Fowler.

Document Revision History

Decision Point for API and Service Implementation Architecture - 9 November 2020

Decision Point for API and Service Implementation Architecture - 21 June 2018

Decision Point for Application Services Implementation Architecture - 11 July 2016

Decision Point for Choosing an Application Services Implementation Architecture - 26 March 2015

Recommended by the Author

Some documents may not be available as part of your current Gartner subscription.

Choosing an API Format: REST Using OpenAPI Specification, GraphQL, gRPC or AsyncAPI

How to Design Great APIs

How to Deliver Sustainable APIs

How to Evaluate API Management Solutions

How to Succeed With Microservices Architecture Using DevOps Practices

Choosing Application Integration Platform Technology

Solution Criteria for Data Integration

Choosing Data-, Event- and Application-Centric Patterns for Integration and Composition

Choosing Event Brokers: The Foundation of Your Event-Driven Architecture

Solution Path for Applying Microservices Architecture Principles