2022 Planning Guide for Application Platforms, Architecture and Integration

Published 11 October 2021 - ID G00753875 - 53 min read

By Analyst(s): Brad Dayley, Gary Olliffe, Kevin Matheny, Richard Watson, Matt Brasier, Steve Deng

Initiatives: Application Architecture and Platforms for Technical Professionals

In 2022, application technical professionals will face a world that requires agile architectures, effective integration patterns, expanded platform capabilities, disciplined delivery practices and versatile skills. This research explains the major trends and associated planning considerations.

Overview

Key Findings

- Application teams are under pressure to deliver more agile application and integration architectures to meet rapidly changing business needs. This situation leads to distributed architectures with simpler components but greatly increased integration complexity.

- The proliferation and diversification of application (and integration) platform capabilities is making it ever-more challenging to maintain skills, source talent, select platforms and design optimal solution architectures.

- When organizations don't invest in governance processes, tools and skills for modern architectures, their applications run a greater risk of failing to meet expectations and deliver business value.

- Skills in agile architecture, APIs, microservices, integration patterns and cloud platform technologies are in high demand. Building skills and creating environments that encourage learning are differentiators for both individuals and organizations.

Recommendations

As an application technical professional responsible for platforms, architecture and integration, you should:
Decompose your application architecture to enable scaling, delivery cadence, technology choice and operational requirements. Then, balance the distribution of data and logic to optimize for specific business needs, delivery cadence, available skills, scaling needs, operations requirements and technology selections.

Ensure that governance helps, rather than hinders, the productivity of your delivery and operations teams by automating service and API governance and embedding it into your delivery pipelines and runtime platforms.

Start with simple REST API-based communications for integration, automation and composition, and then add event-driven or alternate communication models to satisfy clear architectural requirements.

Implement continuous learning by focusing on strong agile architecture, DevOps practices and cloud-native skills first. These foundational skills enable you to make realistic and pragmatic assessments of new architecture and integration technologies, platforms and patterns as they emerge.

Application Platform, Architecture and Integration Trends

Sustainable application delivery is critical to providing business value because it is cost-effective long term and can withstand both evolving business requirements and evolving technologies. However, sustainable delivery requires a deliberate architecture that can continuously adapt to meet new and evolving business needs. This architecture does not happen by accident, and it is not produced by blindly following current trends.

In 2022, successful organizations will refine their application platforms, architecture practices and integration models to adapt new skills, modernize technologies, optimize resources, minimize risk and improve agility. They will build on a strong foundation of proven practices and platform capabilities to further optimize application delivery, making it repeatable, effective and efficient.

The world has witnessed the evolution of microservices, containers, Kubernetes, service meshes, serverless technology, event-driven architecture (EDA), DevOps and continuous integration/continuous delivery (CI/CD) pipelines. A focus on optimization, pragmatism and control will help organizations employ these contemporary technologies more effectively to deliver business value. However a significant challenge remains.
The global COVID-19 pandemic has accelerated digital transformation to enable remote work and to digitalize business workflows that can no longer operate in a traditional fashion. This and other disruptions have forced companies, communities and entire industries to adapt to change in unprecedented circumstances. Application architects must deliver business value in an environment that is increasingly decentralized, fluid and yet connected. In this environment:

- Business processes are enabled by systems that span technical and organizational boundaries.
- Workloads, data and governance are distributed across multiple clouds, data centers and edge computing platforms.
- Application and integration markets continuously evolve. Change is frequently accelerated by open-source solutions, with technologies maturing rapidly but inconsistently.
- Innovation and competition, as well as large- and small-scale disruptions, lead to unpredictable demands for change and integration.
- Integrated, near-real-time flows of information and processes within and between systems are the norm.

Figure 1 shows Gartner’s five planning trends for application platforms, architecture and integration in 2022. They are based on our interactions with clients, the opportunities and challenges they face, and our analysis of the technology markets and ecosystems that support and influence them.
The following technical planning trends are examined in this report:

- **Modern application delivery** will require disciplined distributed architecture.
- **Increasing architectural diversity** will require more than one platform.
- **Automated and integrated governance** will be essential to support productive application composition.
- **Application composition and integration complexity** will require connectivity beyond REST APIs.
- **The need to modernize application architecture** will require evolution of skills development.
Modern Application Delivery Will Require Disciplined Distributed Architecture

Distributed application architecture enables teams to increase delivery agility, build cloud-native applications, adapt to evolving business needs, adopt new technology advancements and create the next generation of user experiences. A distributed application architecture also enables teams to build composite applications from user-facing apps, systems and services that are connected together through API- and event-based communications. Excitement about the potential value of these composite applications is driving a desire for further use of distributed patterns in application architecture.

Distributed application architectures provide significant flexibility, allowing architects to deploy logic and data processing more effectively for better scalability, performance, economy and user experiences. However, distributed application architecture significantly increases the complexity for everyone — from the architect to the security staff to the operations staff. These roles face increased challenges in dealing with distributed communication, identity and access management, security, integration, API management (APIM), data design, and diverse runtime technologies.

Application architects must assess an array of proven architectural approaches to select the appropriate structure and design for an application’s architectural components. For example, mesh app and service architecture (MASA) influences user interfaces, APIs, back-end services and the application platform, whereas web app architecture focuses on the user interface and some API design. Figure 2 illustrates the scope of the most common approaches.
These architectural approaches provide consistency in the structure, design and connectivity of the application components. Each architecture affects multiple components of the overall distributed application, and these effects overlap. For more information, see:

- **MASA: How to Create an Agile Application Architecture With Apps, APIs and Services**
- **How to Apply Design and Architecture to Multiexperience Application Development**
- **Designing Services and Microservices to Maximize Agility**
- **Essential Patterns for Event-Driven and Streaming Architectures**
- **Solution Path for Modernizing Integration Platforms, Architecture and Delivery**
Application architects must become familiar with these architectural approaches, have the required patterns in their toolbox and understand the influence that each approach has on different components of the overall distributed application. Key points to consider are:

- Architectural approaches can overlap or be complementary. For example, to implement a MASA, you may employ web and mobile app architectures for user apps, as well as microservices, serverless and integration architectures.

- Certain mixes of architectural components and approaches are harder than others, and require different levels of expertise, effort and coordination to apply.

- Using the wrong architectural approach (based on the component requirements) results in short-term and long-term problems. For example, using microservices architecture (MSA) to implement the back end of a basic web application that only requires a single service introduces unnecessary complexity, extra effort and long-term maintenance challenges.

- Partnering with enterprise architecture (EA) will help application architects more effectively manage the scope, coordination and governance of their architecture strategy.

- Not all applications will benefit from a distributed architecture.

Application architects should use the following planning considerations when creating distributed application architectures:

- Choose an architectural approach that addresses the organization’s most critical challenges

- Optimize distribution of application components

- Optimize placement of processing logic, state data and communications

- Apply Platform Ops to improve deployment and management

**Planning Considerations**

**Choose an Architectural Approach That Addresses the Organization’s Most Critical Challenges**

The purpose of application architecture is to provide the structure, design principles and prescribed patterns that simplify, and apply consistency to, the implementation and operation of a system.
To select the appropriate architectural approach, you must understand key drivers that define and prioritize distributed architecture concerns. These factors include:

- **Business/organization needs** — Modernization, delivery cadence, innovation, agility, composition, cost reduction, etc.
- **Application requirements** — User experience, device types, data/resources, functionality, performance, reliability, availability, etc.
- **Architectural priorities** — Agility, adaptability, scalability, modernization, integration, deployment, operations, automation, etc.

Understanding each of these will prepare you to address questions about the key distributed architecture concerns shown in Figure 3.

**Figure 3. Key Concerns in Distributed Application Architecture**

<table>
<thead>
<tr>
<th>Key Distributed Application Architecture Concerns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Granularity</strong></td>
</tr>
<tr>
<td>How granular do we decompose apps, APIs and services/systems?</td>
</tr>
<tr>
<td>Apps</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Take the following concerns into account to select the right architectural approach that will best address the key requirements of your distributed, composite and composable applications/systems:

- **Granularity** — *How granularly do we decompose apps, APIs and services/systems?* For app granularity, consider how much to distribute the application requirements over smaller fit-for-purpose app experiences based on the runtime target, device type, interaction modality and channel requirements. For service granularity, consider whether you need microservices or whether domain services or miniservices will suffice.

- **Separation/isolation/abstraction** — *How do we separate data, logic and application state?* Consider how application logic and state data will be distributed. In other words, do they belong on the user device, in an app-specific back-end service/system or in a core back-end service/system? Also consider who will own the data, development and operation of the functionality. Lastly, consider the level of isolation and abstraction required to insulate these components from problematic dependencies.

- **Distribution** — *Where does each of the isolated components run?* Consider where the applications and services will run. Take into account both:
  - *The platform*, such as Kubernetes, serverless, application platform as a service (aPaaS) or virtual machine (VM)
  - *The physical location*, such as an edge device, an edge gateway, a local/regional data center or a central cloud

- **Connectivity** — *How do we connect the components together?* Identify the appropriate mode of connectivity for distributed apps and services. These include REST, asynchronous, gRPC, GraphQL and OData APIs, as well as event-based communication. Also consider how to leverage technologies such as API gateways and service meshes to manage app-to-service and service-to-service communications.

- **Integration** — *How do we make distributed data and logic work together?* Identify whether a data-, event- or application-centric approach is required for the application. Allow the use-case priority, such as data consistency, multistep process or composite application, to drive architectural decisions.
Operability — What are the platform capability and operational requirements?
Identify platform requirements, such as runtime, deployment, automation, error handling, monitoring, security, identity/access and load balancing. Reference those requirements as you select the application platform and design the application operation.

Optimize Distribution of Application Components

Distributing apps, services and other application components adds complexity to your application architecture, implementation and operation. Therefore, only decompose the application into distributed components when the need is dictated by clear business, organizational, application or architectural drivers (as discussed in the previous section). Do not use a microservices approach because it is a “better” architecture. It is not. It's a great way to solve some problems and a remarkably poor way to solve others. You can combine microservices into miniservices when the finer service granularity fails to deliver anticipated benefits — but a better approach is to avoid decomposing the applications in the first place.

A well-executed architecture using distributed application components does provide agility, flexibility, scalability and optimizability, because each component is designed to be built, tested, delivered and run independently of the others. However, the more independent components that exist in your application architecture, the more complex the architecture, implementation choices, deployment effort and operational challenges (such as latency, monitoring, data consistency and versioning). You need to balance these trade-offs and deliver the simplest solution to the problem at hand.

Figure 4 illustrates the trade-off between architectural agility and complexity.
Figure 4. Distributed Architecture Trade-Off Between Agility and Complexity

Because each component in your application architecture amplifies distributed architectural concerns, you should minimize decomposition as much as possible. For example, only decompose services into a microservices architecture when there is a clear need, such as:

- Scaling delivery teams
- Decomposing cognitive complexity
- Allowing independent delivery cadence for different components of the application
- Accommodating alternate technology requirements
- Isolating functionality and data

Optimize Placement of Processing Logic, State Data and Communications
Successful implementation of distributed application architecture hinges on effective placement of processing logic, state management and data, as well as selection of the correct intercomponent communication models. Optimizing those aspects will help define correct separation of concerns, identify data and logic ownership, and avoid unnecessary complexities and dependencies.

Each component in a distributed application architecture has different priorities, usage requirements and constraints for logic, data and communication. For example:

- **Logic** — Identify the appropriate location of application logic. Place logic that enables fast responses to user interactions, device control and local processing closer to user applications. Place logic focused on business processes, domain capabilities and data in back-end systems. Place domain-focused logic within the domain services or systems.

- **State data** — Place application state data on edge devices to support the more volatile requirements of short-term user interactions or immediate device operations. Use application data in back-end services to support stateless APIs, which provide better decoupling from the user apps. Use caching to temporarily store application state and improve performance as necessary.

- **Communication** — Identify the communication constraints for each component in the distributed architecture, based on its placement and connectivity. Use this information to select the optimal communication approaches. Include network, memory and technology constraints when designing application communication to the user experience apps running on remote devices.

As you design your application and integration architecture, identify opportunities to maximize the benefits of applying logic and data processing in the distributed components. Also take the time to identify and implement the proper communication technologies, such as API gateways and event brokers, to enable communication between the different components. The following documents provide additional discussion around API and event communication design:

- **How to Design Great APIs**
- **Applying Event-Driven Architecture to Modern Application Delivery Use Cases**
- **Choosing Event Brokers: The Foundation of Your Event-Driven Architecture**
Apply Platform Ops to Improve Deployment and Management

Seek to simplify deployment and reduce the impact of operational complexity in your distributed applications. Applying a Platform Ops strategy abstracts some of the complexity by implementing a self-service platform. Select and support platforms that simplify the delivery and operation of the applications and enable the organization to scale DevOps practices across multiple independent software delivery teams.

The goal of Platform Ops is to simplify the lives of development teams and enable them to effectively deliver and operate applications — not the infrastructure stack on which those applications run. The platform team should:

- Establish a cross-functional team that has the platform and operation skills to support distributed architecture platforms and technologies.
- Identify repeatable delivery and operational tasks, automate them, and then leverage this automation to scale tasks across multiple teams.
- Treat the platform as a product. The platform team should collaborate with the platform's customers (the application teams) to evaluate and identify new capabilities that can become part of the platform.

Using Platform Ops to Scale and Accelerate DevOps Adoption provides more details about Platform Ops, how it applies to application delivery, and how to effectively implement it.

Increasing Architectural Diversity Will Require More Than One Platform

In last year's Planning Guide, Gartner cautioned that fascination with Kubernetes would distract architects from optimizing platform selection. The triumph of marketing Kubernetes has made it foremost in architects’ minds, and it has become synonymous with cloud-native architecture. However, many effective platforms for application delivery are available, not just Kubernetes. Teams must be pragmatic about selecting the platform that best matches their application requirements, skill level, and deployment and operations needs.

Moreover, teams should choose a platform focused on productivity at the architecture layers that are primary to their chosen application architecture. Different platforms will support productivity to varying degrees at these architecture hot spots, and productivity is heavily based on utilizing existing skills. Thus, different people will find different platforms more or less productive. Figure 2 above illustrates the hot-spot areas of significant impact and complexity for each example architecture.
Another observation we’ve made, based on thousands of client interactions, is that organizations often strive to standardize on a single platform.

The search for a single platform to rule them all is fruitless.

Successful enterprises realize that they will maintain multiple platforms into the future. This approach enables pragmatic platform selections for each use case or family of apps that the enterprise needs to build or modernize. Gartner advises clients to navigate between two extremes:

- Pursuing shiny new platforms, regardless of business value delivered
- Sticking with existing platforms to avoid the need to modernize legacy applications

As an illustration of platform diversity, Figure 5 shows a spectrum of cloud platform styles with some of their characteristics: intended audience, use case, deployment model and programming model. Choosing platforms is a multifactor decision of course, but teams should keep the linear relationship between flexibility, control, complexity and productivity in the spectrum diagram top of mind.

More complex distributed architectures need more complex and flexible platforms. Simpler architectures can run on complex platforms, but the overhead of these platforms makes doing so a poor decision. For example, consider a basic web application. Can you run it on Kubernetes? Absolutely. Should you? No, because it doesn't need the power of a distributed platform, and because the operational cost of the platform is higher than simply hosting the web app using a VM and automation code.
To optimize application platform selection, architects should follow these planning considerations:

- Decouple application architecture from platform selection to maximize flexibility
- Select an application platform based on teams’ development and operations needs
- Don't make portability the primary reason for using Kubernetes
- Leverage PaaS technologies and cloud services rather than building and maintaining platforms in-house

Planning Considerations

Decouple Application Architecture From Platform Selection to Maximize Flexibility
Let the architecture and design of your applications drive your choice of application platform and how you use it — not the reverse.

Working from the platform up to define your application architecture is an anti-pattern that will cause you to compromise on how easy, fast or effective the application is to deliver and support. By contrast, starting with application architecture reduces the risk of introducing dependencies that increase the challenge of long-term support.

Your application architecture can also be defined at the same level of abstraction as an API. It should not make assumptions about the availability of specific cloud provider services, Kubernetes primitives, a serverless model, VM instances or any other infrastructure construct. Avoiding dependencies on specific infrastructure gives you some degree of portability, but that is not the primary goal. The real goal is to allow extension and evolution of the platform over time. To maximize flexibility, your applications and services should not care whether they are deployed to specific platforms. The platform-specific configuration should be external to your code, with environment data, storage, networking and security managed through configuration manifests that are late-bound to your platform.

Software architects are rethinking their approach to application delivery. Packaging and deploying applications in containers enables architects to adopt new approaches to application delivery, resiliency and scalability. But in the same way that application software developers should not design their solutions to work with a specific VM hypervisor, they should also minimize their solutions’ dependence on Kubernetes’ native constructs, configuration and capabilities. Containerized software deployment using Kubernetes brings new abstractions that can resolve old challenges. However, Kubernetes also introduces new complications in configuration, networking, security, storage and operations.

Although your platform for modern cloud- or container-based application delivery should provide a flexible environment to support a variety of architecture, design and communication patterns, there will inevitably be scenarios where the available platforms constrain your design. Look at these scenarios as opportunities to evolve the platform. However, only make this investment if the value that can be delivered by removing the constraint outweighs the additional cost, effort and complexity of evolving your platform.
Figure 6 shows Gartner’s core reference architecture for the components of a cloud- or container-based application platform that supports modern service-centric (microservices) development practices and architectures. It also shows additional components that are needed to support or optimize the platform for different application delivery personas, scenarios and use cases.

**Figure 6. Extended Platform Capabilities for Modern Application Delivery**

Select an Application Platform Based on Teams’ Development and Operations Needs

Your platform fits your application architecture, but does it support your desired software delivery processes?
Gartner frequently speaks to organizations that have established new containerized, cloud-native platforms, but are struggling to get development teams to use them. The problem is not that the organization has chosen the wrong platform. It is that this choice was made without the participation of development teams.

Development teams are tasked with delivering business value by getting features into production with low effort, low risk and low cycle times. Giving them access to a Kubernetes cluster is solving a small fraction of the problem. Changing platforms is a significant effort for development teams, and they will resist it unless that platform makes things easier for them. Don't make the mistake of investing years into building a platform before talking to developers about it. Your imagination about what developers want is a poor substitute for talking to them.

Ensure that your platform capabilities match your organization's current (or intended) delivery model, rather than forcing application teams to adapt to bleeding-edge solutions. For example, are you aiming to accelerate or optimize a traditional model, or to implement a continuous distributed delivery model? Figure 7 contrasts a traditional model that consolidates updates into “release trains” against a dynamic (and complex) approach that decouples deployment of discrete components into “release taxis.”
Figure 7. Define the Development and Operations Capabilities You Need Based on Your Target Operating Model

**Define the Dev and Ops Capabilities You Need Based on Your Target Operating Model**

Traditional vs. Continuous Delivery Release Cycles

---

### Traditional Release Cycles

- **Development**
  - Code
  - Debug
  - Commit

- **Application Releases**
  - Build
  - Test
  - Validate
  - Deploy
  - Operate

- **Isolated Feedback**

### Continuous Delivery Release Cycles

- **DevOps**
  - Service
  - Team A: A, B, C
  - Team B: D, E, F
  - Team C: G, H, I
  - Team D: J, K, L

- **Application Releases**

- **Continuous Feedback**

---

Source: Gartner

728856_C

---

The platform will need to:

- Support the developers’ “code-debug-commit” inner loop
- Contain an integrated “build-test-deploy” pipeline
- Provide multiple levels of monitoring, tracing and analytics to provide insight into the health of the applications and to support the DevOps feedback loop

You will also need to take the following aspects into account:

- Application runtime and dependency maintenance
When selecting application platform technologies, focus on the features your teams need to deliver business value, not the technology trend of the day.

**Don’t Make Portability the Primary Reason for Using Kubernetes**

Many organizations have multicloud environments. Some of these environments emerge for deliberate strategic reasons (e.g., diversification for cost arbitrage, or risk mitigation). Others result from tactical decision making (e.g., developer convenience or access to new cloud-specific capabilities). However, many multicloud environments arise accidentally (e.g., an acquired company has a different cloud standard). When planning how to target application delivery in these situations, organizations often have a desire to enable consistency and portability between environments.

Ensure that there is business value tied directly to this portability vision. For example, the flexibility to quickly redeploy or extend an application may help mitigate significant business continuity risk or enable support for customers that need to be served from specific providers or geographies. In general, it is better to have an effective exit strategy to deal with cloud vendor lock-in than to invest in portability for a particular application to mitigate provider risk. For more details on this, see *How to Create a Public Cloud Integrated IaaS and PaaS Exit Plan* and *Designing a Public Cloud Exit Strategy*.

In addition to seeing Kubernetes as a means to cloud independence, Gartner clients commonly view it as a way to support strategic or transitional hybrid cloud computing platforms and/or distributed cloud and edge computing scenarios. Using the open-source distribution of Kubernetes also brings the promise of application portability between distributions and platforms. Building for portability is an expensive insurance policy that is rarely claimed against, and the claims process is tortuous in any case. Adopt Kubernetes primarily for its benefits of agility, standards and consistent control of distributed systems. Portability is a great side effect, but don't make it the primary driver for your Kubernetes implementations. (See *Assessing Kubernetes for Hybrid and Multicloud Application Portability*.)
Although most cloud providers offer Kubernetes-based services, there are still many operational and functional differences between these services. Just standardizing on Kubernetes as a platform is not sufficient to provide portability between cloud providers. This point is evidenced by the relative strengths and weaknesses we've highlighted in the following research series:

- Solution Scorecard for Amazon Elastic Kubernetes Service
- Solution Scorecard for Google Kubernetes Engine
- Solution Scorecard for Microsoft Azure Kubernetes Service

Due to this diversity, you must balance three trade-offs when working toward multicloud consistency: the level of platform consistency required, the dependency on a single vendor solution, and the level of effort to support and operate the platform. Another key trade-off is that your application architecture cannot leverage any other platform services (e.g., database as a service [DBaaS] or event broker as a service) that are not available everywhere. Typically, you will have to deploy and manage your own services to get consistency. This will add further complexity and cost.

The other critical factor is consistency and portability of operational practices. You might invest in building your applications to be deployable to a variety of container services by developing and testing dedicated manifests to optimize networking and storage for different Kubernetes environments. However, the operational tools, processes and practices you use in each environment will be different. Investing in third-party cloud management and automation tools can alleviate some of this inconsistency, but that abstraction means investing in another platform component with associated cost and complexity. (See Solution Criteria for Cloud Management Tools and Solution Path for Infrastructure Automation). Examples of the differences you will need to consider to ensure multicloud operational consistency or portability are disparate identity, networking and storage policy models, APIs, and CLIs. The deployment and management processes you automate for one environment will not be portable to others — nor will your development and operations skills and experience.

**Leverage PaaS Technologies and Cloud Services Rather Than Building and Maintaining Platforms In-House**
Building and sustaining a complete, productive, secure and performant PaaS for application delivery comes with a significant upfront build investment and long-term maintenance commitment. This is true whether you base your platform on a portfolio of public cloud infrastructure services or build a private cloud platform. Leave do it yourself (DIY) platform building to those that will get a return on that investment by spreading the cost across customers and partners (i.e., technology vendors, service providers and enterprises with extensive developer ecosystems).

Kubernetes may turn out to be the best match, or the underpinnings of the best PaaS solution, but it’s an incomplete application platform in isolation, in need of opinionated customization and packaging. Kubernetes vendors distinguish themselves in the market by addressing PaaS requirements for application teams.

Unless you’re in business to deliver a PaaS, don’t create your own from building blocks like Kubernetes. Instead, evaluate platform products and services that deliver the platform capabilities you need, typically as interoperable or composable elements. In all cases, building or sourcing a platform is just the start. There will still be ongoing platform management, integration and operations. Your goal as a platform provider to application teams is to choose products that reduce their burden of repetitive, automatable tasks and enable them to get on with delivering software into production.

As a platform framework, Kubernetes is at its best and most powerful when used as the “platform for building platforms.” However, the adoption and visibility of Kubernetes in the marketplace will lead architects and organizations to obsess about the technology rather than the capabilities their organizations and applications need to fulfill their business objectives.

In the six years since it reached version 1.0, Kubernetes has become a focal point for the cloud technology marketplace. Generated from a Gartner dataset, Figure 8 captures dependencies in the cloud market and demonstrates just how prevalent Kubernetes is as a basis for cloud platforms, deployment targets or integration targets.
In Figure 8, nodes in the graph represent cloud platform products and services. The size of the nodes is proportional to the number of inbound links. Different product regions of the ecosystem are represented by the colors, as shown in the legend. The links between nodes represent a relationship (e.g., “A is built using B” or “C runs on D”). Based on the high degree of interconnectedness, one can conclude that vendors in the cloud platform ecosystem are betting heavily on Kubernetes.

However, this dominance Kubernetes has in the market largely derives from its use by platform providers and vendors. For application architects and developers, Kubernetes alone falls far short of a complete application platform — by design. It provides a flexible and extensible distributed execution environment, but lacks many abstractions and capabilities to streamline the development, delivery and operation of modern distributed application software, such as software supply chains. Providing those abstractions and capabilities is the role of the platform ecosystem. To turn Kubernetes from a platform framework into an application platform, you will also need to deploy, operate and support many additional capabilities for the life of the platform. These include:

- Ingress control, or an API gateway, to manage and mediate inbound requests
A service mesh to manage, secure and monitor service-to-service communication

Backing services for persistence and messaging

Monitoring and observability tools

Security tools to verify and protect your containers, networking and cluster configuration

CI/CD and platform automation tools

For further details, see Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications.

Application, platform and infrastructure architects must collaborate to determine the most effective way to source and support these capabilities and align them to the needs of their current and planned application workloads. They should focus on the needs of the application development teams as consumers of the platform and balance those needs against platform operational complexity (see Using Platform Ops to Scale and Accelerate DevOps Adoption). Application architects should not avoid Kubernetes; rather, their teams should consume it in the most appropriate form for their application, and in moderation compared to other available platforms. In many cases, that will mean that only the organization’s platform provider or Platform Ops team needs to know whether and how it uses Kubernetes. Other teams should be platform consumers and should be protected from leaky abstractions over Kubernetes APIs.

Automated and Integrated Governance Will Be Essential to Support Productive Application Composition

Enterprises and the systems and processes that they operate are becoming ever-more “composite.” Multiple factors are driving a combination of distribution and composition:

- Increasing use of SaaS
- Creating and consuming internal APIs, API-based products and B2B APIs
- Building and operating cloud-native applications
- Decomposing applications using microservices architecture principles
- Composing applications using mesh app and service architecture
- Creating multiexperience apps
Gartner uses the term “composable business” to describe how an enterprise itself can be organized to include flexibly composable business capabilities. Doing so allows the organization to respond more quickly to customer or stakeholder needs and provide new services or solutions faster. This operating model also extends into relationships with partners, suppliers and customers. Both internally and externally, these capabilities and the links between them are enabled by technology. See Future of Applications: Delivering the Composable Enterprise for further details.

The technical composition patterns to support composable business are varied:

- **API-centric architecture** exposes shareable business capabilities and data.
- **Low-code application platforms** enable streamlined composition of new user experiences.
- **Integration and automation** technologies compose processes and aggregate technical components into business capabilities.
- **Microservices architecture** decomposes the delivery and operations of components and services.
- **Mesh app and service architecture** composes applications from components and services.

The increased flexibility and agility that these patterns offer comes at the cost of increased complexity, which requires more governance.

---

Traditional application governance practices were established when systems were slower-moving and when change was viewed as a risk, not an opportunity.
By contrast, modern composite enterprises and systems thrive on adaptability, so they need governance that manages risk without inhibiting change. Some of the drivers of composition have established governance patterns, such as using APIM to govern APIs. Others, however, have minimal governance that is often manual, slow and an inhibitor to delivery agility.

This planning trend focuses on incorporating productive automated governance as part of your application and integration modernization activities:

- **Productive** governance helps the organization extract value from a resource by making it easier and safer to both deliver and use.

- **Automated** governance ensures control and consistency, with minimal human intervention and minimal delays to the participants.

Productive automated governance focuses on guiding good behavior using guardrails, rather than preventing and then approving or rejecting actions based on rules or controls. With this approach, in the normal course of delivery, any minimal and occasional impediment added by governance is offset by significant benefit from reduction in risk.

The planning considerations for 2022 related to this trend are:

- Choose patterns based on fitness, not hype
- Focus governance effort on enabling controlled efficiency with guardrails
- Integrate and automate governance controls to enable self-service and autonomy
- Prioritize governing interfaces

**Planning Considerations**

**Choose Patterns Based on Fitness, Not Hype**
The amount of buzz surrounding patterns such as MSA and EDA is driving organizations to choose these patterns without regard to how well they fit the actual needs of the organization. Vendors and consultants play into the hype, and the idea that certain patterns are uniformly better and more modern has taken hold. The reality is that architectural patterns vary widely in terms of their suitability for different circumstances. MSA is a powerful and effective pattern for increasing the flexibility and scalability of an application, but it dramatically increases the operational complexity of that application. Applying a pattern because it is in style, rather than because it is the right tool for the job, is a recipe for disaster.

The key “hot” application architecture patterns that we see continuing to dominate architecture discussions and planning are:

- **Microservices architecture** — Use MSA when you are creating applications that demand a fast pace of change. Ensure that the increased cost of operation is justified by the ability to deliver business-driven changes quickly, and that the demand for change will remain over time. Do not use MSA for applications with a slow pace of change. See How to Succeed With Microservices Architecture Using DevOps Practices for information on the prerequisites for success with MSA.

- **Event-driven architecture** — Use EDA when you need to decouple data providers from consumers, and when you need to enable multiple consumers of a single stream of events. Do not use EDA for cases where the sender needs to know whether a message has been received. See Choosing Event Brokers: The Foundation of Your Event-Driven Architecture for information on implementing event-driven architecture.

- **Mesh app and service architecture** — Use MASA when you need to create multiple user experiences from shared services. Do not use it for cases where a single N-tiered application will suffice. See MASA: How to Create an Agile Application Architecture With Apps, APIs and Services for guidance on this pattern and how to apply it.

- **Headless/API-centric architecture** — Use headless architecture when you need to decouple user interfaces from back-end services in order to modify existing capabilities or integrate new capabilities. Do not use it when operational simplicity is a higher priority than agility and flexibility. See Accelerate Digital Transformation With an API-Centric (Headless) Architecture for Enterprise Applications and How to Control Integration Delivery Costs and Risk Using Tactical Integration Practices for guidance on using this pattern.
Focus Governance Effort on Enabling Controlled Efficiency With Guardrails

Governance is often a “dirty word” among technologists. It usually conjures up images of forms, stage gates, approval boards and heavyweight processes to gather information that is never used. But as described above, governance can be an enabler if implemented properly, and doing so will be essential in the dynamic, distributed and flexible business organizations that are striving to become composable.

Implementing productive automated governance should enable change rather than constrain it. Such governance should focus on letting work happen productively and safely rather than preventing it.

Governance that focuses solely on compliance prevents work from being done, causing workers to avoid or subvert it.

A good example of governance that is both automated and productive is embedding security controls into a CI/CD pipeline. Automation comes from static code analysis, dynamic analysis of automated test cases, registration of artifacts and collection of metadata. Each of these is automatically executed for every deliverable passing through the CI/CD pipeline. Productivity comes from the development teams’ ability to engineer their software to a known quality dictated by the tests, without being inhibited by “security approval” delays for every change. Focusing on business outcomes (delivering a secure application) instead of just process execution allows security teams to concentrate on optimizing the necessary security checks and audit trail. They can use the data collected from the CI/CD process to assess and respond to risks and incidents. This approach eliminates the need for “fire drills” to approve last-minute changes manually and improves productivity for the security team.

Governance of application and integration platforms and delivery should smooth and accelerate the path for “good” solutions and behaviors.

Integrate and Automate Governance Controls to Enable Self-Service and Autonomy
Governance processes and practices that are disconnected, manual activities separated from the work they intend to guide will result in lower adoption and diminished value. Manual governance steps are easier to skip, are more prone to human error or intentional brevity, and result in a governance view that drifts from reality. A common, though largely historic, example of this in application and integration architecture is service-oriented architecture (SOA) governance. As organizations adopted SOA in the 2000s, they experienced many of the complexities identified in this planning trend — discrete service development, dependencies between services, proliferation of component services and a desire for reusability. The prevalent registry/repository model of SOA used at the time was rarely successful, for several reasons:

- It relied on service designers and developers to manually register interfaces, service definitions and metadata.
- The tools used were largely disconnected from the runtime environment.
- Usage of the data in the registries and repositories was limited at best.

Other than compliance, there was little motivation for the service developer to follow the process. Service consumers searching for artifacts and metadata either failed to find them or found data that drifted from reality. In the end, SOA registry repositories became full of decaying service definitions, and SOA governance in general was discontinued.

The goals of SOA governance were sound, but the implementations were neither productive nor automated. As the discipline of SOA governance has evolved into API management, APIs (of all types) have become a critical component of digital business architecture, and integration governance has grown in importance. However, the approach has changed. APIM is a good example of productive automated governance. Well-implemented APIM makes it easier for service developers and data owners to publish APIs with consistent authentication, effective authorization, and robust runtime consumption controls that protect their capacity and keep track of users and usage. At the same time, APIM provides API consumers with self-service access to well-defined API contracts, known policy limits and, often, immediate approval to access a resource. As a result, both API providers and consumers become more productive.
The same principle applies to application security, master data access, event streams or cloud platform consumption: Focus your governance approach on the needs of the consumers of the resource.

Design and integrate governance controls into the platforms and tools that your organization uses to deliver value, and reduce the human burden of protecting the organization (through governance).

Governance is largely about collecting data to inform decisions that reduce complexity or minimize risk. Figure 9 uses fleets of aircraft as a metaphor for software system delivery. Ungoverned delivery is efficient when a small fleet of aircraft is operating in a large area, but as soon as there is a danger of overlap, the pilots need data about their neighbors. That insight can come from preplanning the changes (i.e., filing a flight plan that is checked by other pilots or controllers). However, this approach is obviously fragile and typically works best at a macro level, where strategic decisions are made for groups of aircraft. For maximum agility and in-flight decision making, each aircraft needs accurate, up-to-date information from onboard systems and controllers on the ground.
Figure 9. Focus on Governance That Supports Safe, Autonomous In-Flight Decision Making

**Focus on Governance That Supports Safe, Autonomous In-Flight Decision Making**

- Apps, Services and Applications
- Observation/ Monitoring
- Context/ Information
- Localized Decisions
- Shared Decisions

**Ungoverned**
- Dangerous Maneuvering

**Preflight Governed**
- Coordinated Maneuvering

**In-Flight Governed**
- Informed Maneuvering

By integrating the governance controls into your teams’ delivery processes, you can help them deliver value and help your organization keep control of its complexity. This pattern applies whether it is API publication and runtime API policy enforcement, event-stream discovery and access, or data catalog publication and data access.

**Prioritize Governing Interfaces**

Connection points provide visibility into the workings of increasingly distributed enterprise systems and processes. Knowing what is connected, how, why and by whom is fundamental to informing future decisions.
In modern application and integration architectures composed of distributed services, events and data, you should focus governance efforts on the interfaces exposed by those resources. Whether it's a REST API, a GraphQL API, a set of events and topics, or a database connection, the interface defines both the contract expected by consuming components and the scope of exposed resources. Governance of the resources used to implement and fulfill the interfaces can be decoupled from the implementation and operation of the interface.

By applying a productive automated governance approach to the interfaces, you can retain visibility of the dependencies in the system and across systems. By governing access to resources at the interface, you can track who is using them at implementation time — including who or what has been granted access to integrate with, develop against or otherwise access the resources. You can also track who is using the resources at runtime. This tracking covers which consumer apps, processes or services are actually using the interface over time; which versions of the API they are using; what capacity they are using; and what their usage pattern is.

For REST, GraphQL, SOAP and other request-response-style APIs, governance means implementing API management practices and capabilities. Use these to ensure that access to your services is authenticated, authorized and mediated by an API gateway that can enforce access policies, track usage and abstract underlying complexity (e.g., MSA or application integration and process orchestration) from consumers. See How to Design Great APIs, How to Deliver Sustainable APIs and How to Successfully Implement API Management for further information.

For event-based interfaces, apply the principles of API management, but recognize that the tooling in this market is still emerging. Certain vendors, such as IBM and Software AG, have added some support for event-based interfaces to their API management platforms. Other vendors, such as Axual, Lenses.io and Solace, provide tooling focused specifically on helping to govern event streams and event-driven systems by integrating more directly with your event broker middleware. See Choosing Event Brokers: The Foundation of Your Event-Driven Architecture for further information.

For data resources, look for data governance tools and platforms that offer self-service and ease of use to consumers while enabling data owners to control, trace and monitor use of corporate data. See the 2022 Planning Guide for Data Management for further details.
Application Composition and Integration Complexity Will Require Connectivity Beyond REST APIs

The need to adapt to change, support evolving customer expectations and keep up with technology advancements demands a composable architecture based on components that can be assembled, reassembled and extended as required. In modern architecture, these components are typically delivered as services, which provide APIs and produce or consume events, but also occasionally extend to support user experiences.

Establishing a composable architecture allows application architects to innovate and adapt to the changing needs of your business. It can also help deliver composition capabilities to other roles, including fusion teams, integrators and developers.

The interfaces and communication models that connect the components of an architecture are key to enabling composability, maintainability and flexibility.

The long-term shifts from point-to-point connections, to multipurpose SOAP interfaces, to consumer-centric REST APIs have come with significant improvements in how enterprises define, discover, monitor and manage these connections. However, REST APIs only deliver one of the interaction styles required by a modern business. Enterprises must augment their use of REST interfaces with additional connectivity options, such as event streams and GraphQL, to provide the optimal integration points required by specific composition scenarios.

Enabling a mixture of interface types provides flexibility for application developers and allows them to implement optimal communication between disparate components in the enterprise’s ecosystem. By applying strong API design principles to all interface types, application architects can more clearly define how components of a system can integrate with each other. Disciplined use of interface contracts employing structured specifications – such as asynchronous APIs, gRPC/protocol buffers, GraphQL schema and OpenAPI Specification (for REST) – is key to applying API design and life cycle management to your interfaces. Additionally, sourcing or upgrading API design tools, portals, gateways and related middleware with support for these additional interface styles will make it easier to monitor, maintain and manage the interfaces correctly.
Application architects should use the following planning considerations to augment their existing REST APIs with additional communication models:

- Use MASA to establish the discrete components of the architecture
- Establish an API-led approach to integration, automation and composition
- Use event-driven APIs to minimize coupling and increase agility
- Augment REST APIs with GraphQL to increase dexterity

Planning Considerations

Use MASA to Establish the Discrete Components of the Architecture

Application architecture has shifted away from large, siloed, monolithic applications toward applications composed from multiple fit-for-purpose apps, designed with multigrained service architecture and connected via mediated APIs.

Mesh app and service architecture, shown in Figure 10, allows you to optimize for agility through modularity at all levels of a system. It promotes clear definitions of the components involved, the data and functional requirements of those components, and the optimal communication channel requirements between components. MASA enables the composable business. For more information about MASA, see MASA: How to Create an Agile Application Architecture With Apps, APIs and Services and Use MASA to Deliver an Agile Multiexperience Enterprise Application Architecture.
The following are some key principles that MASA prescribes in building such an architecture:

- Each multiexperience app, API and multigrained service should be independently built, tested and deployed.

- APIs connecting apps to back-end services (outer APIs) should be mediated to promote consistency, simplify management and provide abstraction.

- APIs should apply a consumer-centric design principle, where the developers/applications using those APIs are the consumers.

- A domain-driven design should be used to identify strong boundaries, and to establish integration and composition requirements between those domains.
Establish an API-Led Approach to Integration, Automation and Composition

Both integration and composition are heavily dependent on APIs, but their goals are different. Integration views applications and data as islands to be connected. It builds bridges to enable or ease the flow of processes and data, and well-designed APIs can simplify the task. By contrast, a modern composable-application architecture focuses on delivering systems to meet changing business requirements by creating systems of components.

Integration uses APIs to connect systems, automation uses APIs to execute processes and composition uses APIs to create systems.

The optimal approach to integration and composition starts with establishing effective APIs. The key to success with APIs is instituting a consumer-centric approach to interface contract design. How to Design Great APIs describes a consumer-centric approach to API design that should be combined with API delivery and management (see Figure 11).
To create effective interfaces that are long-lived, valuable and maintainable, take the time to:

- Use personas, storyboards, contracts and documentation to engage the consumers of your APIs throughout the conception, planning, design, implementation and operation phases of the interface life cycle.
- Clearly define the capabilities needed to support the integration and composition requirements of your API consumers.
Use Event-Driven APIs to Minimize Coupling and Increase Agility

An event-based communication approach supports integration requirements that will benefit from its higher responsiveness, flexibility and scalability. Event-based communications are part of a broader event-driven architecture and support open, scalable, streaming, near-real-time and push-based communication models. Develop skills in EDA to ensure you understand how it works and which communication use cases will benefit from event-based communication.

Event-based communication differs from request-response communication in that the consumer of the event processes the event asynchronously from the producer of the event. Therefore, events should only be used when the application or integration can tolerate an eventual-consistency and asynchronous model.

Effective application architecture in a composable business requires a combination of both API styles and communication patterns. Request-response and event-driven communication are not mutually exclusive, and each has its own strengths. The key to successfully applying both is to understand the use cases where events are the optimal approach. Essential Patterns for Event-Driven and Streaming Architectures provides guidance on many of the EDA patterns and when to apply them. Event-driven API contracts can be specified using a variety of techniques, including asynchronous APIs, GraphQL subscriptions and OpenAPI Specification 3.1 Webhooks.

Identify the optimal communication approach based on the integration and composition requirements. For example, ask questions such as:

- Is the integration transactional or query-based?
- What is the required transfer rate?
- What data must be shared?
- How flexible does the interface need to be?

Select the API style (REST API, event, GraphQL, asynchronous, gRPC) that most effectively supports those requirements.

Describe the interface using a contract-driven approach, where the contract defines the API or event channel based on the consuming-application needs, provider capabilities and technology constraints.
The foundation of event-driven communication is an event broker, which enables some components to publish events to an event channel and others to subscribe to those events. The event broker enables connectivity, enforces event delivery semantics, and provides capabilities to deploy and manage the event interface. Selecting and implementing an appropriate event broker is the key to successfully deploying event-based communications. Choosing Event Brokers: The Foundation of Your Event-Driven Architecture describes the key capabilities and decision factors for selecting an event broker.

**Augment REST APIs With GraphQL to Increase Dexterity**

Some of the common challenges Gartner clients relay about their traditional REST APIs stem from the difficulty of anticipating future needs in the design of the APIs. Examples of these challenges include managing API versioning and dealing with limited API applicability (i.e., APIs that are useful for one client app but not another). GraphQL shows significant promise in providing more flexibility to the API. It enables developers to use query language markup to define what data to return or actions to take on the back end. This markup capability essentially allows application developers to customize the API access to the needs of their application.

GraphQL is a good companion to REST APIs when your application developers require more dexterity in the way they access back-end resources and when they have the expertise to understand the implications of how they are accessing data. However, your organization must be willing to learn the GraphQL fundamentals and tolerate the access, security and traffic impact of GraphQL on your application architecture, API gateways and API management. Assessing GraphQL for Modern API Delivery provides an analysis of GraphQL and describes its key benefits and weaknesses.

As shown in Figure 12, a GraphQL implementation has three main components that you must consider:

- **GraphQL client** — You will need to implement a GraphQL framework as part of your client app architecture. Additionally, the client app developers must learn the fundamentals of GraphQL and clearly understand the impact that their queries will have on back-end systems.

- **GraphQL service** — You will need to implement a GraphQL server and define an effective schema. The schema is key because it is the interface for the client apps. Additionally, you will need to take the time to configure identity, access and security policies, as well as connect to your existing API gateway/management technologies.
The Need to Modernize Application Architecture Will Require Evolution of Skills Development

Multiple factors are driving enterprises to modernize their application architecture. Embracing digital business, reducing costs, meeting new user expectations and exploring new opportunities are only some of these factors. Modernizing application architecture represents significant challenges, and application architects and developers will need to embrace new architectural approaches, apply new delivery approaches, support new platforms and adopt new technologies. This will require not just new skills, but also a new approach to learning and enhancing skills.
The organizations that are best-equipped to both acquire the skills they need and continue to refine them are those that have a culture of continuous learning. A culture of continuous learning is one that values acquisition and improvement of skills. It supports those objectives in multiple ways, making various activities available for individuals to increase their skills (see Figure 13). Learning activities are initiated and led by the application architects, developers and other technical professionals, but the organization must also support and sponsor them with time, leadership opportunities, money and other methods.

**Figure 13. Continuous Learning Cycle**

**Continuous Learning Cycle**

- Lunch and Learns
- Innovation Sprints
- Paired Programming
- POC/Prototypes
- Online Courses
- Book Clubs
- Communities of Practice
- Code Kata

- Apply Agile Learning Principles
- Adopt a Versatile Mindset
- Prioritize Foundational Skills
- Align with Business Value
- Identify Valuable Outcomes

**Continuous Learning Activities**
- Initiated and lead by individuals
- Encouraged and sponsored by organization

**Apply Valuable Outcomes**
- Practical: code libraries, POCs, configurations, blueprints, architectures, designs
- Growth: Mentoring, best practices, confidence, trust, technical capabilities
- Productivity: Improved delivery cadence, lower technical debt, reduced bug count, automations
- Product: Improved UX, new functionality, better performance, accurate testing, increased workforce productivity

Proven Practices to Enhance Skills in Application Architecture, Development and Software Engineering provides additional insights and best practices for developing a culture of continuous learning.

Application architecture and integration professionals should find ways to contribute to a culture of continuous learning by participating in available activities and taking the initiative to create, start or encourage activities. Such activities may include:
Planning Considerations

Develop Agile Architecture, DevOps and Cloud-Native Skills

No single individual, team or organization can learn all the application architecture and development skills that exist today. Thus, you should focus on foundational skills based on proven practices and technologies. Foundational skills also provide a significant positive impact on the individual, team and product.

For example, the following skills are foundational because they enable flexibility in your architecture, application delivery and platform choices:

- **Agile architecture** — This discipline involves defining a composite architecture to create applications comprising independent apps, APIs and services. An example is the mesh app and service architecture (see MASA: How to Create an Agile Application Architecture With Apps, APIs and Services). This discipline also includes the ability to modernize an existing architecture by decomposing a monolith and encapsulating its functionality within APIs to build independently deliverable and scalable services.
DevOps — This discipline begins with strong agile practices and includes the skills necessary to implement and operate an effective tool chain that supports CI/CD. DevOps skills also include practice-based capabilities, such as container management, API mediation and performance monitoring. DevOps removes existing process barriers and replaces them with a more consistent process discipline focused on continuous improvement.

Cloud-native application architecture — This discipline includes skills to design, architect, integrate and operate applications that are ready to deploy and run in a cloud environment. It requires knowledge of the principles and design patterns that enable applications to fully utilize the agility, scalability, resilience and elasticity provided by cloud computing. It also requires an understanding of the capabilities provided by cloud platforms and how to apply them to simplify application implementation, delivery and operation.

You should prioritize which skills to acquire or enhance based on the skill’s ability to provide:

- **Architectural agility** — Does the skill enable delivery of components of the application (such as individual apps, APIs and services) at independent cadences and using optimal technologies?

- **Productivity at scale** — Does the skill allow you to design, architect and implement the application in a more productive way, and does that productivity scale across the team and organization?

- **Innovation** — Does the skill allow you to be innovative with your architecture and integration designs, use optimal technologies, and apply optimal architectural models, patterns and practices?

- **Improved user/customer experience** — Does the skill improve the UI or make the application more performant and reliable in ways that will help users be more productive or create unique/engaging digital experiences?

Figure 14 from *Essential Skills for Application Architecture* categorizes application architecture skills and evaluates their positive impact, learning curve and impedance to apply.
Prioritize Skills Based on Business Value and Application Needs, Not Hype

A common trap that application architecture and integration professionals fall into is selecting skills to acquire based on current trends and industry hype, rather than on the capabilities that will help them deliver more value faster. The previous section discussed prioritizing skills that are fundamental. You must also take a look at additional skills based on the real impact to you, your applications and your organization. You can do this by asking the following questions:

- Is this a skill that has already been identified by my organization as critical?
- Is this a skill that we are currently lacking on my team or within my organization?
- Can I apply this skill in my current role?
- Will this skill enable me to expand my current role in the products I am working on?
You are looking for strong and clear “yes” answers to these questions. Avoid spending significant time learning skills that provide uncertain answers to these questions.

Apply Practical Learning Activities Against Real-World Problems to Gain Knowledge and Experience

The key to effectively learning application architecture skills is understanding the learning curve. The learning curve has two axes — knowledge and experience — as shown in Figure 15. Successfully learning skills means balancing those two components. Notice that gaining too much knowledge is dangerous, because you know how to do things without the experience to understand when and why to do them. At the same time, having to learn a skill solely through experience is painful.

**Figure 15. Skills Learning Curve: Knowledge vs. Experience**

In most cases, people learn better from someone who has greater knowledge and experience than they do. However, that is not always an option, and in the absence of someone to learn from, you should find others to learn with.
Depending on the skill and your individual learning style, there are many learning activities in which to partake, such as tutorials, online courses, bootcamps, conferences, POC or prototype builds, book clubs, and lunch-and-learns. Look for those opportunities within your organization. If they don’t exist, take the initiative and start one. Prioritize practical learning activities that enable you to gain both knowledge and experience. Some examples include:

- Communities of practice
- POCs or prototypes
- Pairing (programming, design, architecture)
- Team rotation or sharing

These types of activities provide multiple benefits. They enable you to focus on real-world problems, they integrate into your normal workflow (so you don’t need to set aside learning time), and the outcomes are often used in production applications.

The following documents provide insight into specific learning activities for specific skill sets:

- Proven Practices to Enhance Skills in Application Architecture, Development and Software Engineering
- Essential Skills for Modern Application Development
- Essential Skills for Application Architecture
- Essential Skills for Modern Integration Architecture
- How to Build Successful Communities of Practice for Knowledge Management
- Assessing Online Learning Platforms for Technical Skills Development

**Define Practical, Measurable Outcomes for Learning Activities**
The most successful organizations encourage and support a culture of continuous learning and give technology professionals the time and opportunities to learn new skills. However, application architects often run into the barrier of justifying the time and effort to learn new skills either to themselves or to their management. An effective method to overcome this challenge is to define valuable and measurable outcomes from learning activities.

As you identify learning objectives, also define one or more specific outcomes that clearly have value to you, your team, the products you deliver and your organization. Additionally, these outcomes should be measurable in some way that allows you to demonstrate your progress to others. Some examples of these activity outcomes include scaffolding, prototypes, reference architectures, design guides, configurations, code libraries, best practices and training materials.

**Dedicate Time to Develop Soft Skills as Well as Technical Skills**

As an application architecture and integration professional, you should find ways to contribute to a culture of continuous learning. That means participating in available activities and taking the initiative to create, start or encourage activities. Although many soft skills, such as time management and listening, will help you achieve your professional goals, you should prioritize the soft skills that will most contribute to building a successful culture of continuous learning.

**Document Revision History**

- 2021 Planning Guide for Application Platforms, Architecture and Integration - 9 October 2020
- 2020 Planning Guide for Application Platforms, Architecture and Integration - 7 October 2019
- 2019 Planning Guide for Application Platforms and Architecture - 5 October 2018

**Recommended by the Authors**

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

- Proven Practices to Enhance Skills in Application Architecture, Development and Software Engineering
- How to Design Great APIs
- MASA: How to Create an Agile Application Architecture With Apps, APIs and Services