workato

WHITEPAPER

iPaaS Architecture Deep Dive

Why the path forward is Cloud-Native vs. Cloud-Optimized

In this paper

Introduction

Integration requirements, and associated complexity, have evolved from the basic need for connectivity: “how can I connect my on-prem CRM with my on-prem DB?” to “how can I integrate the plethora of SaaS applications and in-house applications while automating all the steps associated with any given business process that crosses over these systems in a consistent, reliable, maintainable, and scalable way?”

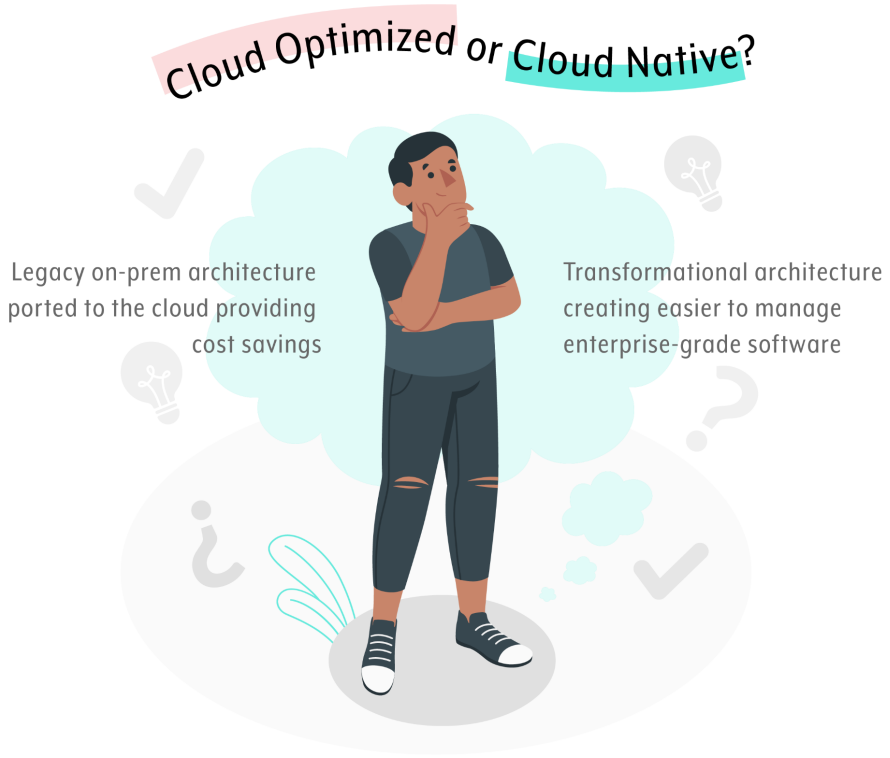
This increasing complexity usually adds a demoralizing overhead on integration developers who not only need to worry about solving a business problem but also need to focus on the non-functional requirements that any solution needs in order to be considered “enterprise-grade” (e.g. reliability, traceability, error handling, observability, etc.).

To address these new and ongoing challenges, and through the convergence of existing integration technologies and the cloud, the Integration Platform-as-a-Service (iPaaS) market emerged. However, “how” each iPaaS vendor decided to adopt the cloud has limited the potential value of their platforms.

Fully leveraging the cloud and understanding its possibilities is usually what the industry calls “cloud-native”. In this paper, we’ll discuss why cloud-native is critical for an iPaaS platform.

But before digging into why cloud-native is the answer to commoditizing the ever-growing integration complexity, let's level set on what cloud-native means.

Cloud-native vs. cloud-optimized



The best way to understand “cloud-native” is by comparing it with its close cousin: "cloud-optimized".

As the cloud became mainstream, organizations from every industry decided to look at the cloud from one of two perspectives: either as a cost-saving mechanism or as an opportunity for disruption and transformation. This is both from the perspective of how they leverage the cloud for internal IT operations and also how they leverage the cloud in shaping the product or service they deliver to customers.

In the former (and most common) group, organizations are drawn to the cloud as a way to save money on CapEx and operational costs. To achieve this goal, the focus is on porting their existing on-prem loads to the cloud as close to their "as-is" state as possible to minimize disruption and risk.

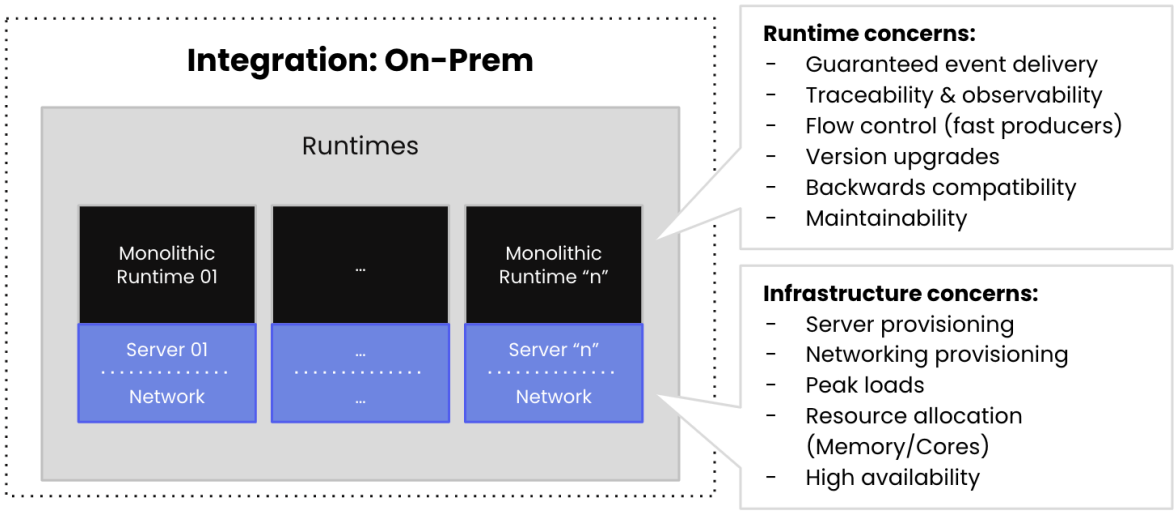
On the other hand, a smaller set of organizations saw the cloud as a way to reinvent themselves and/or their respective markets by realizing the full potential of the cloud: speed, scalability, resiliency, and agility. Netflix and Zoom are prime examples of cloud-native products and their potential for market disruption.

> While there is no standard definition of cloud-native, there is consensus that it is the result of building modern applications for the cloud by following the principles of "Microservices", "Containerization" and "Container Orchestration".

iPaaS: Why cloud-native matters?

Out of all the different applications in the cloud-native paradigm, this article focuses on the benefits that a cloud-native iPaaS has over cloud-optimized iPaaS platforms.

The following diagram describes the different concerns that on-prem integration platforms imposed on their users before the cloud emerged:



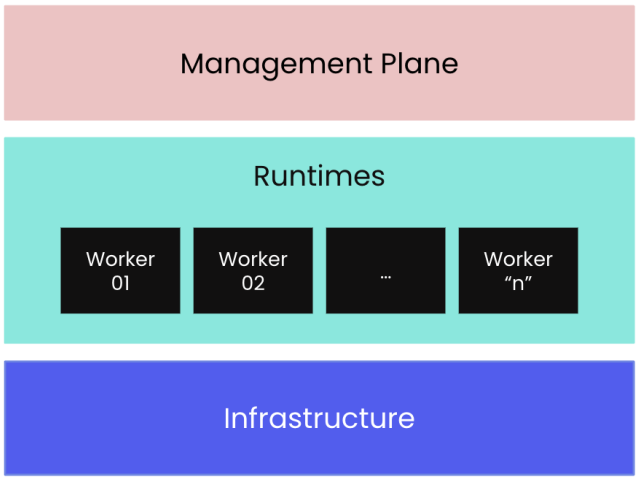
As the cloud became the new Northstar, a new integration space was created: iPaaS. However, the cloud architectural paradigm employed as a guiding principle defines the potential impact and benefits of such platforms for the customers leveraging them.

Isn't iPaaS a standard architectural principle?

The short answer is **no**.

As mentioned earlier, the iPaaS space is the result of continuous evolution that emerged from the need for connectivity. The technological underpinnings of these new platforms were directly affected by how each vendor decided to embrace the cloud to "build their iPaaS alternative".

At a high level, we can generally agree that every iPaaS has 3 main logical components:



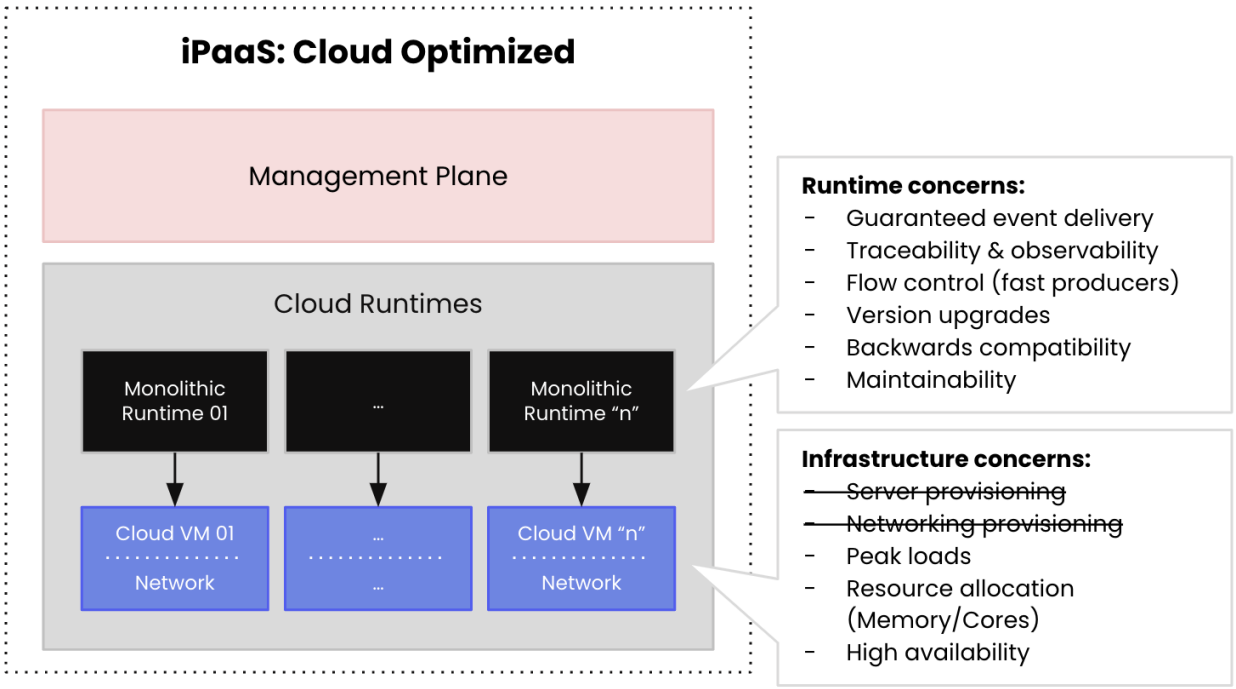
1. **Management plane:** Usually, the main UI access point to manage user access controls, application deployment, runtime provisioning, and orchestration (not the user interface for building integrations & automations)
2. **Runtimes:** Where the rubber meets the road. Execution engines for processing the integration/automation job
3. **Infrastructure:** Underlying resources that support both the management plane and runtime provisioning

Understanding these basic elements is key because most vendors have focused on building their cloud management planes with microservices in mind, but with complete disregard for leveraging microservices for the runtimes that manage the heavy lifting processing of their platforms.

Furthermore, these runtimes have preserved their inherent monolithic architecture from when they ran solely on-prem (a.k.a. cloud-optimized approach or in some areas similar to a lift-and-shift approach). Regardless of where they run today, whether it be on-prem, cloud, containers, etc., at their core, most of them are still self-contained monoliths.

Why a cloud-optimized iPaaS isn't enough

A lot of this has to do with the timing of when the iPaaS product was initially developed. If an iPaaS product was developed during the first wave of the cloud (often defined as pre-2010), then it's almost guaranteed to have a cloud-optimized architecture.



By following a cloud-optimized approach, the only concerns that get removed are:

- Server provisioning
- Networking provisioning

While these are great benefits over the on-prem architectures that came before, all traditional limitations associated with integration platforms remain a customer concern. Customers still need to worry about non-functional requirements such as reliability, traceability, version upgrades, peak loads, high availability, etc., thus hindering the real value that can be derived from a true cloud-native iPaaS.

What a cloud-native iPaaS should look like

With the understanding of how most “cloud-optimized” iPaaS platforms are architected, let’s explore what a cloud-native iPaaS should look like and, more importantly, the value it must bring to organizations.

For this purpose, we are going to look at an iPaaS platform through the lens of:

- Microservices-based architecture
- Containerization and container orchestration

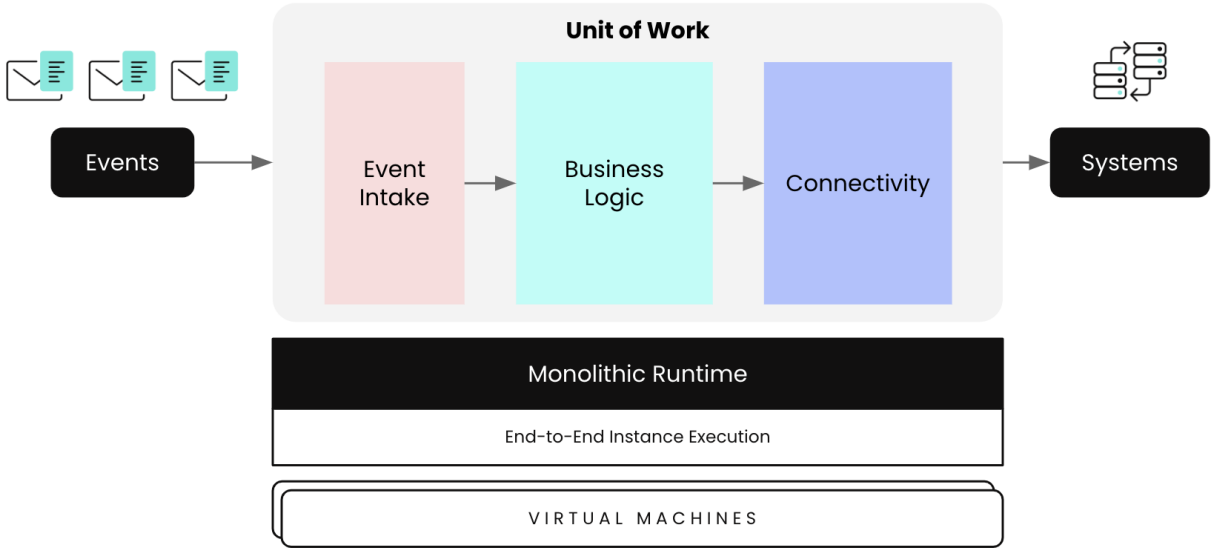
Management plane

Building a management plane based on a microservices architecture is a no-brainer. As with virtually any other modern web application, there are clear-cutting concerns you would typically encapsulate in microservices, such as security, logging, auditing, as well as UI components. Since microservices in this context are pretty much aligned with most of the content out there on the web, I won't delve into specifics. But when in doubt, you can always count on Martin Fowler's *definition* of microservices.

For this article, we are assuming a microservices-based management plane is the norm and not the exception.

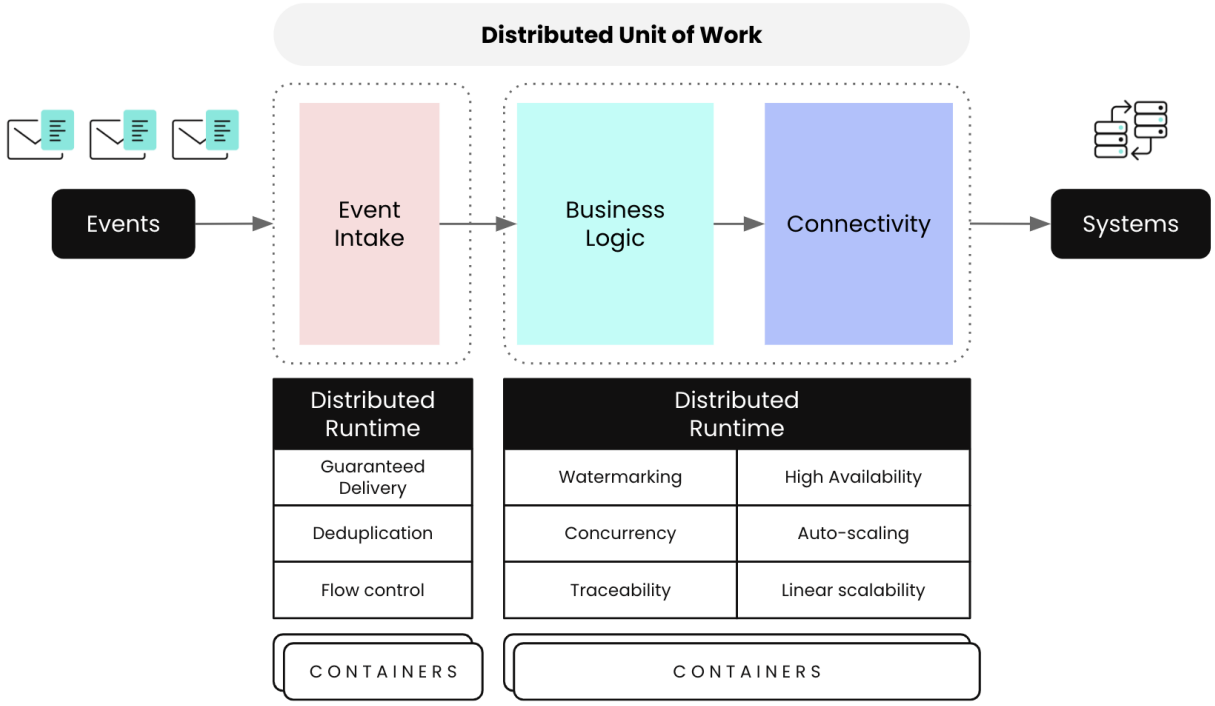
Runtimes

Those who have used other integration platforms will know that an iPaaS runtime usually executes a unit of work (UOW), such as a “flow” or a “process” end-to-end. These UOWs are typically self-contained integration logic that includes information about how to receive an event (understanding “event” as a broader concept that can include synchronous API requests, async messages, etc.), what to do with it (e.g. business logic) and where to send it to (e.g. target destinations):



In a traditional "cloud-optimized" iPaaS platform, a monolithic runtime will execute a given instance of a unit of work end-to-end on top of a single virtual machine (or equivalent). Furthermore, because these runtimes typically favor stateless architectures, by default, the execution of these instances is ephemeral, which usually translates to a lack of message delivery guarantees, high availability, and traceability, among many other capabilities. Depending upon the nature of the UOW, it can also lead to performance issues.

But what if your runtimes followed a microservices architecture and your "units of work" became distributed rather than monolithic?



A microservices-based runtime architecture would allow for capabilities such as "separation of concerns", where you could potentially have a different, purpose-specific type of runtime capturing events from any source system or application (e.g. different runtimes for API requests, async messages, webhooks, etc.) focusing on the following (and more):

- **Guaranteed delivery:** To ensure captured messages will be processed no matter what
- **Deduplication:** To ensure events captured are unique
- **Flow control:** To ensure that "fast event producers" won't overwhelm "slower event consumers"

On the other hand, you could have another set of purpose-specific runtimes that are only focused on executing the business logic associated with each unit of work with a different set of concerns, such as:

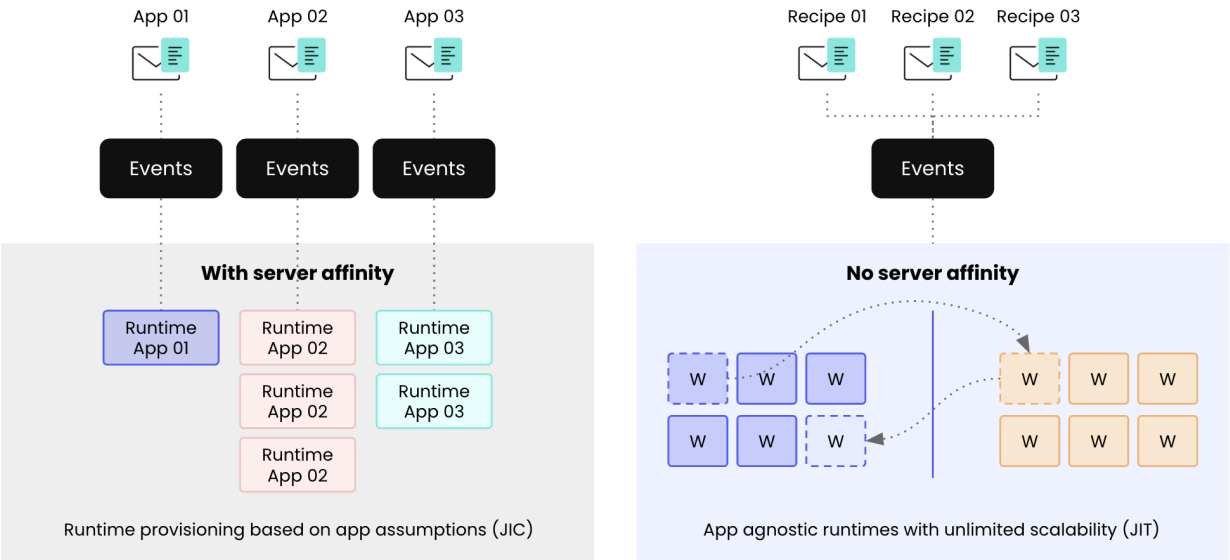
- **Traceability**: To have transaction-level visibility every step of the way
- **Auto-scaling**: To ensure there will always be capacity available to execute each unit of work without delay
- **Linear scalability:** To ensure execution will scale gracefully regardless of peak loads

Infrastructure

While microservices define your solution architecture, your container and container-orchestration strategy define your deployment architecture. Docker and Kubernetes are arguably the industry standards when it comes to cloud-native initiatives.

However, cloud-native is not just about running containers, but leveraging cloud technologies to their maximum potential. At the end of the day, anything can run on a container but using containers to change an architectural paradigm is what promotes change and disruption.

Why is avoiding server affinity key to scalability and zero-ops?



In a traditional model, vendors have the concept of an application (e.g. piece of code) and a runtime (e.g. server). Once you finish building an application, you deploy it to a particular runtime. This creates a tightly coupled server affinity between the application and the runtime running it. While this model may give you the perception of "full control", in reality, it transfers all the operational burden associated with managing those runtimes to the person who is building the application. Things like sizing, peak load estimation, high availability, and performance, become commonplace concerns in this type of model.

Conversely, by avoiding server affinity, a cloud-native iPaaS can have an application-agnostic container infrastructure that is ready to pick up events on-demand from any "application".

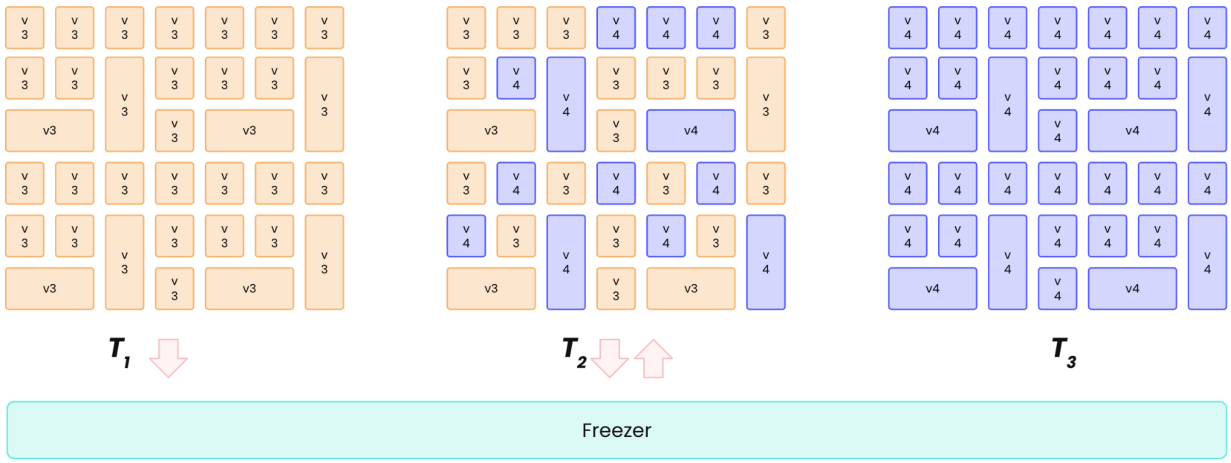
Similar to the concepts of just-in-case (JIC) and just-in-time (JIT) from the manufacturing world, the notion of pre-allocating servers or cores to a given application can resemble the strategy of preemptively assigning resources in case they are necessary (JIC) which potentially incur in waste and oversizing. In contrast, dynamically allocating generic runtimes to any type of event from any type of

application on-demand, resembles the modern process of allocating resources efficiently when they are needed (JIT).

As a result, a true cloud-native iPaaS can benefit from an auto-scalable infrastructure that is constantly monitoring the availability of "idle workers" to ensure the timely execution of every application event.

Following this approach with traditional models would be cost-prohibitive as you would have to scale app-specific runtimes on their own at the risk of causing initialization delays (e.g. to spin up new containers, deploy new applications, provision networking access, etc.) and wasted resources.

The benefits of avoiding server affinity don't stop there. The ability to have the execution runtimes completely decoupled from the builder's intent (e.g. application) enables zero downtime and zero impact upgrades.



The image above is a visual representation of leveraging this architecture to upgrade runtimes from V3 to V4 without requiring downtime.

This level of indirection empowers product teams with new levels of agility as they can enhance and build new capabilities into the platform without impacting existing applications.

Therefore, this unique architecture enables zero-ops as all the platform infrastructure management overhead becomes a platform concern while letting end users focus on delivering value.

Now that we understand what a cloud-native iPaaS should look like, let's explore how Workato not only embraces this architectural paradigm to its fullest but also delivers new levels of value and productivity.

How we've architected Workato to be cloud-native

Self-Documenting

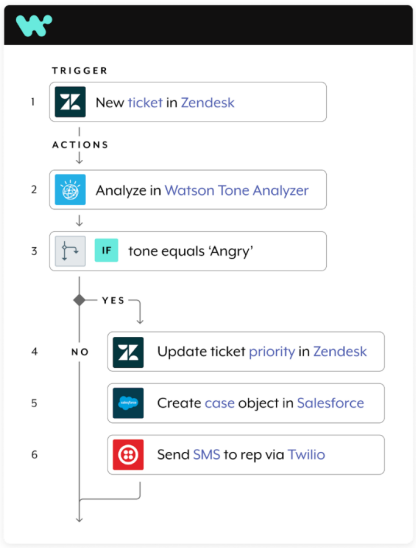
Easy-to-read flows written in plain English that all can understand and maintain

Powerful Logic

Linear | Branching | Looping | Try/catch | Error handling

Connect to Anything

Cloud | On-prem | Micro-services | Custom Connector SDK



Event Based

Real Time | Scheduled | Batch

Smart Connectors

1400+ apps | Auto-dedupe | In-Sequence Delivery | No data loss

Low-code, Enterprise-grade

Community Recipe templates | Instant Deployment | Auto-scaling | End to End Data Encryption in Transit or at Rest | Data masking | Data retention control | Activity audit log with streaming

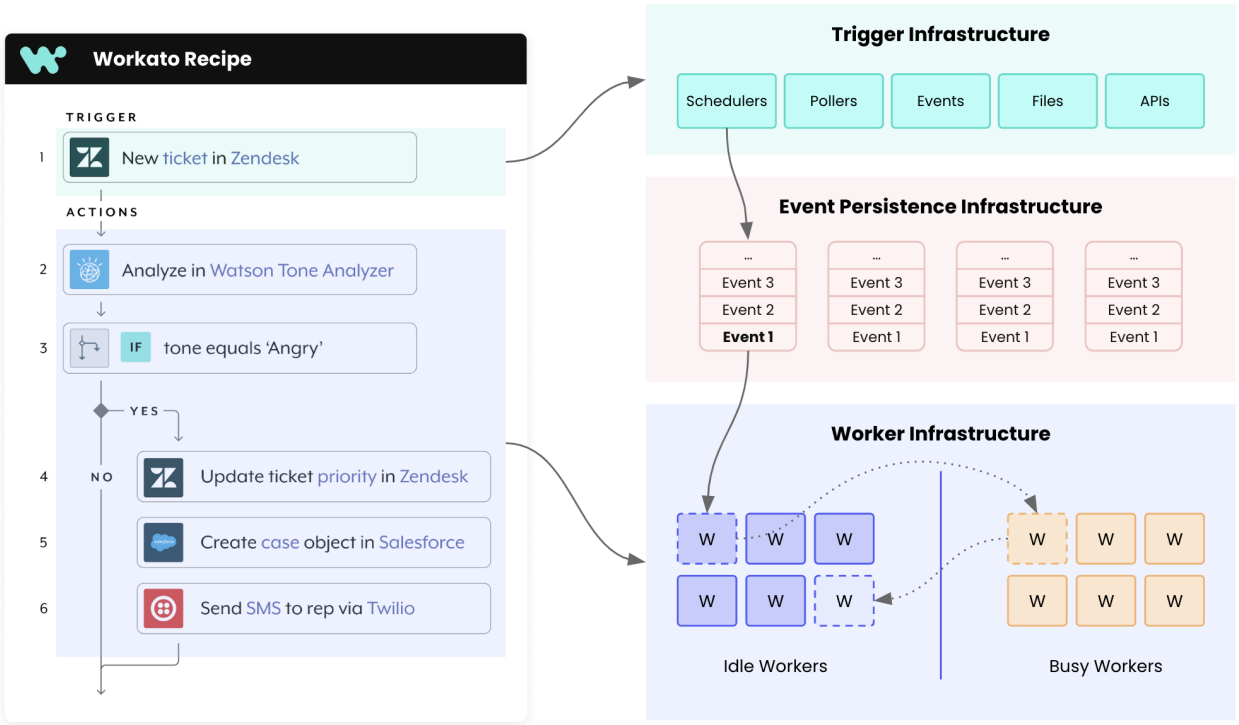
Workato is a cloud-native iPaaS that in many ways behaves more like a "Serverless Integration and Automation Platform" than a traditional iPaaS platform. This is a direct result of Workato's platform architecture and how it fully embraces the cloud-native paradigm.

The Workato Recipe (an example is provided in the image above) is the main building block for integrations and automations. From a logical standpoint, it may be seen as the equivalent of flows or processes in other platforms as it's where builders will configure which events to listen to, what to do with them, and where to send them after. But that's where the similarities end.

Basic recipe structure:

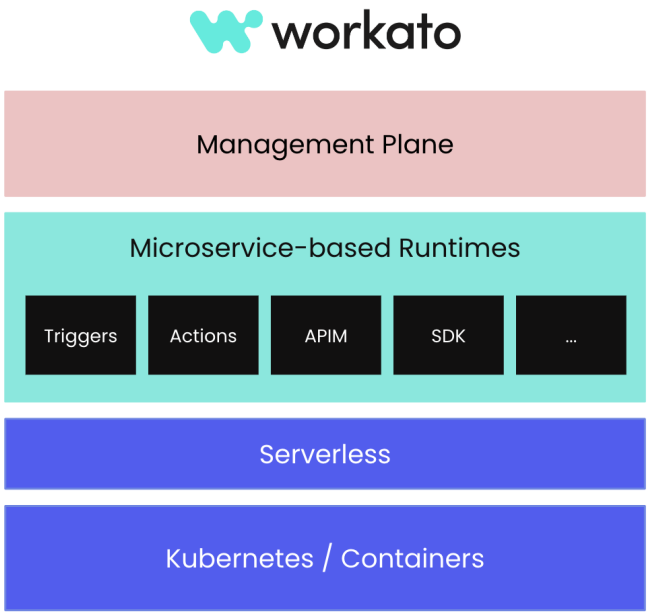
- **Triggers:** Handle incoming events (scheduler, events, pollers, webhooks, APIs)
- **Actions:** Steps to perform once an event is received

At first glance, it may feel like you are configuring a single unit of work that will be executed as a whole just like any other runtime. However, the way we execute recipes is completely different.



- Recipes are descriptors of the "builder's intent" and are decoupled from our execution runtime
- Recipe logic is evaluated on-demand during execution by any available (idle) server, thus removing the notion of a “deployed app” and closely aligning with a serverless execution paradigm
- While recipes look like a single set of instructions, Workato has separate auto-scalable container infrastructures dedicated to triggers that are constantly listening to events
- Once these events are captured, they are transcribed (persisted) into our highly distributed, and highly performant queuing infrastructure (without users even realizing it)
- As events are loaded in our queues, we have yet another auto-scalable container infrastructure dedicated to picking up these events and executing the set of actions defined on each recipe

This is one example of how our runtime is broken into microservices that can be independently maintained and scaled as needed to support our entire customer base in a multi-tenant way. Along those lines, every other major component in our platform follows this architecture:



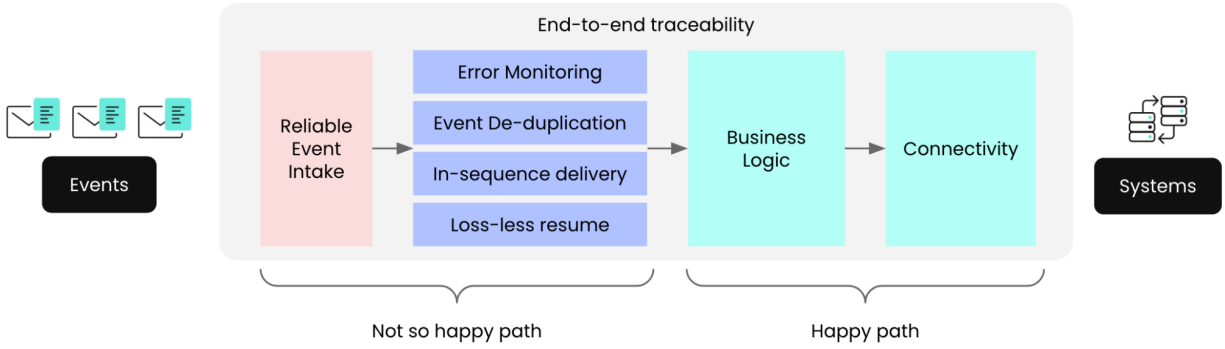
- Triggers
- Distributed queues
- Actions
- APIs
- Webhooks
- Workbots

The benefits of a cloud-native architecture for customers

Workato's architecture not only removes integration complexity but also allows organizations to focus on solving their business problems. In the next section, we'll discuss in detail the value of Workato as a cloud-native iPaaS.

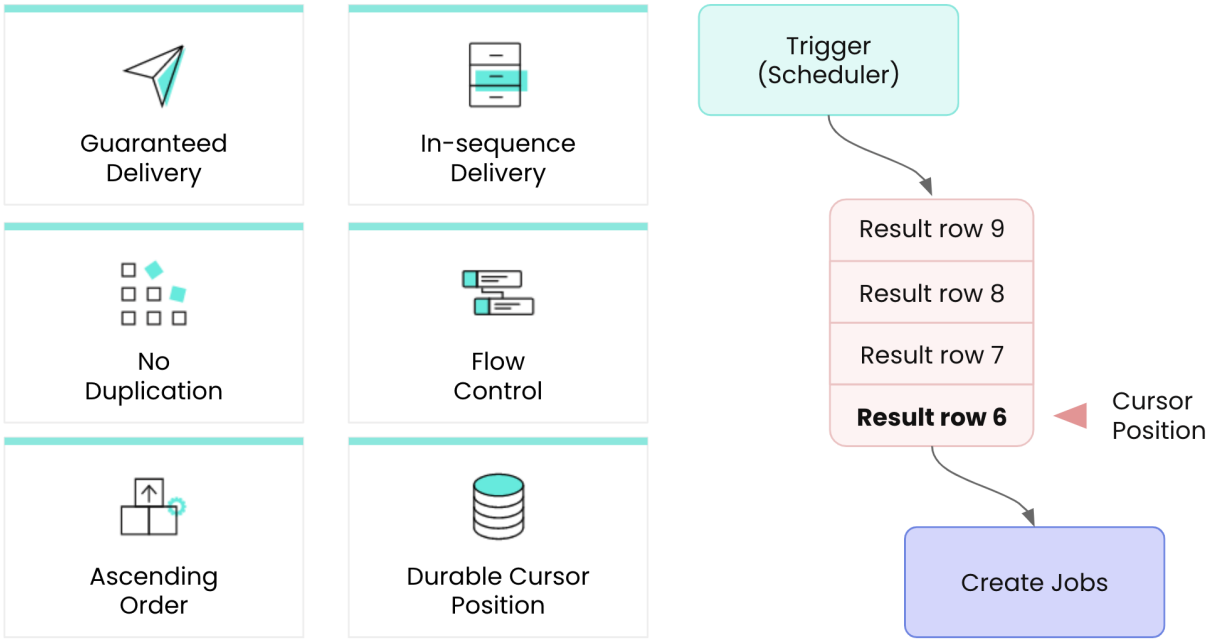
Remove the common integration boilerplate

With a traditional iPaaS, the time and effort to build key non-functional requirements (e.g. in-sequence delivery and traceability of records) is arguably as much as building the business logic itself. Time is the most precious resource in any business so any opportunity to win back time to focus on what's driving the most value for the business is imperative.



In legacy platforms, conservatively speaking, about 50% of the final solution logic (and time spent building it) is dedicated to implementing these repetitive, cumbersome (but highly necessary) requirements to deliver true enterprise-grade solutions (scalable, fault-tolerant, etc.).

What makes Workato so unique is the fact that by building a cloud-native microservices-based platform, most of these requirements are not only supported but also provided by default out-of-the-box to our customers, whether they realize it or not.



- Guaranteed delivery
- Message persistency
- Error handling
- Transaction retries
- In-sequence delivery
- Deduplication
- Watermarking

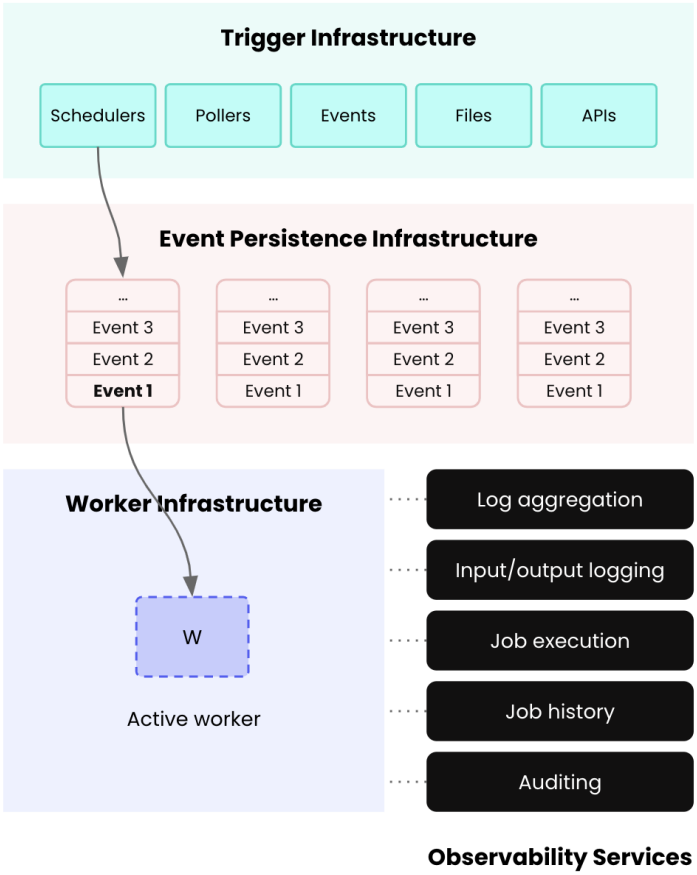
These are a few of the capabilities that, in Workato, are no longer a developer's concern but a platform concern. We not only increase the overall quality of the solutions being built but also increase developers' productivity by allowing them to focus on real business problems.

Troubleshooting made easy

Yet another example of how microservices and cloud-native can make our customers' lives better is when we are operating live in production and something bad happens.

For the longest time, logging and observability have arguably stayed at the bottom of the "cool tech pyramid" (if there was such a thing). Developers usually not only dread adding logs across their solutions but also frequently lack proper standards or practices to do so. Needless to say, by the time someone realizes the logging capabilities are not enough, something would go wrong, and sadly there is no way to know what happened (without spending excessive amounts of time and resources that is...).

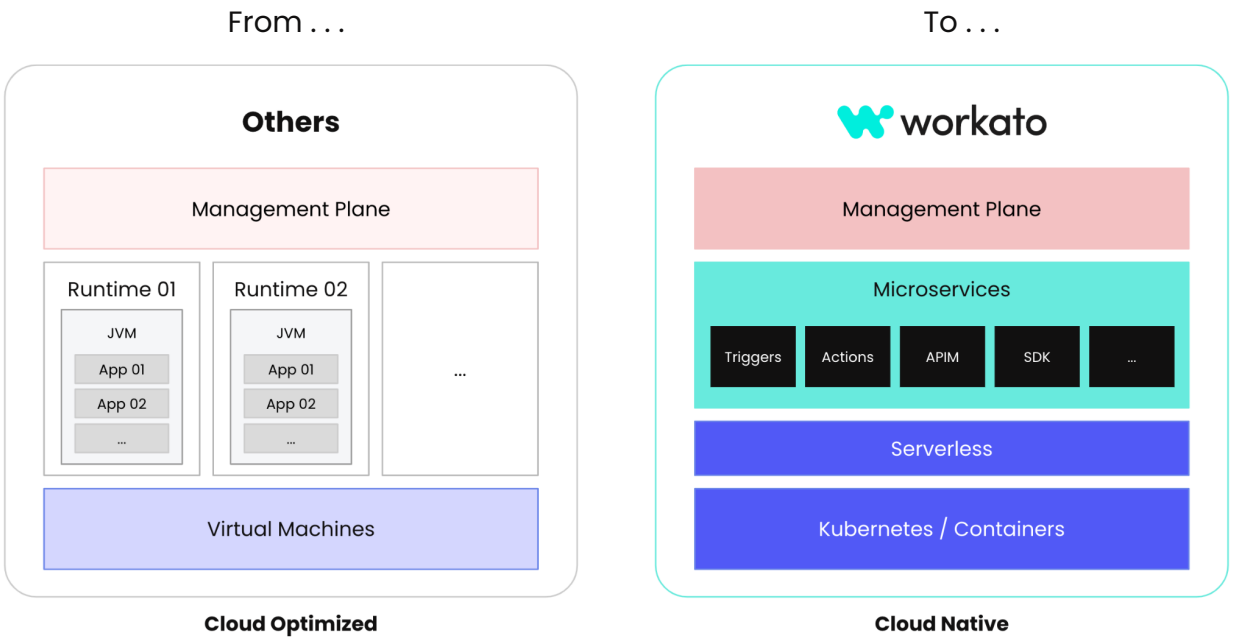
> What if traceability was a platform and not a developer concern? What if you could always tell what happened with every single transaction at every step? What if you could even turn on debugging at will and retry the specific transaction you are interested in? All of that is precisely what Workato can offer because of (as you may have guessed by now) the way we are architected.



- Out-of-the-box transaction-level traceability
- Out-of-the-box auditing
- Job history and job retries
- Zero downtime upgrades

Out-of-the-box, every recipe execution is enhanced by our observability services, which provide all the necessary information for triage and troubleshooting if something goes wrong.

Conclusion



To sum up, the integration and automation space has greatly evolved over the years, and **Workato's platform is driving innovation in this market forward.**

We are entering an era where cloud-native applications are becoming more pervasive by focusing on delivering value and the integration and automation space is no exception.

Hopefully, this article helped explain why, as opposed to monolithic runtimes of the past, cloud-native can help commoditize and abstract integration complexity and enable our customers to focus on delivering value.