Java in cloud computing

Moving Java workloads to cloud environments with Red Hat

For more than 25 years, Java™ has been a highly popular programming language—and Java remains one of the top-3 most-used languages for enterprise software development, even today. Countless business-critical systems around the world have been built using this iconic language.

When Java 2 Platform, Enterprise Edition (J2EE™) 1.2 was introduced in 1999, it marked the birth of enterprise Java and helped shape the future of software development. Major versions have included Java 1.3, Java 1.5, Java 8, and Java 11. Over the years, many people have proclaimed "Java is Dead," but Java is still going strong with the recent release of Java 18.

Java has always been well suited for enterprise environments because it is object-oriented, platform-independent, stable, secure, and backwards compatible. In addition, Java has offered a well-defined set of APIs and extensive developer tooling since its inception. Java is also well loved because it is easy to learn, and the promise to run anywhere has seen Java adoption surpass billions of devices (e.g., servers, computers, smartphones, game consoles)—all in all, JVM is fast, stable, and a mature language that caters to businesses worldwide.

In the last decade, however, application development has undergone a critical shift, as developers move away from traditional, monolithic architecture patterns to lighter and more modular services and functions built using cloud computing and Kubernetes. That's all to take advantage of a more efficient, distributed cloud environment. Where does this leave Java, which was not originally designed for fast-boot-up times, low-memory footprints, or containers? Java has had to evolve, and if developers want to enjoy the benefits of Java in cloud environments, they should look no further than Java itself. Let's take a look at some of the design assumptions and the changes that make Java an effective choice for cloud computing and most importantly for the enterprise applications footprint.

Design assumptions

Is Java ready for cloud environments? It was originally built on certain design assumptions that make it incompatible with cloud computing and Kubernetes. Due to the way application development has evolved in the last decade, these design assumptions are no longer valid.

These are some of those design assumptions:

Long-running applications

Java was designed so applications could work anywhere and for a long time, maximizing throughput at the expense of footprint. This made Java ideal for large datacenters with maximum memory and CPU, hosting a large number of applications on one server. Due to the nature of the vertical stack, this made Java optimal due to both cost and performance. While this particular style of deployments was effective for a vertical stack back then, it of course poses challenges to Java applications moving to cloud envi-

ronments, which are more distributed and inherently feature elastic scaling. Thus, applications that were built as large monoliths also end up being more resource hungry to the underlying stack, and with long release cycles, have also accumulated technical debt along the way making them "sticky" to the stack they were built upon.

Java has been very effective with long-running applications. Let's take a look at the two core features that enable long-running applications to perform at scale.

Garbage collection

Garbage collection (GC), an automated memory management feature in the JVM, was needed in the past for applications that run for a long time and are never shut down. GC periodically removes objects in memory that are no longer in use, so as to reclaim memory for other needs.

Although serving an important purpose, this feature also poses a downside in that applications can slow down, pause or even stall during the GC process, because of the massive CPU load required. In a mission-critical application, these latency and availability issues could cause business transactions to be interrupted or the user experience to be degraded.

While essential at the time, GC is no longer relevant because most applications in the cloud are smaller, and designed to be quickly turned off and on, and scaled up and down. Applications in the cloud usually run for milliseconds and then are shut down, so they do not run long enough for garbage collection to be necessary.

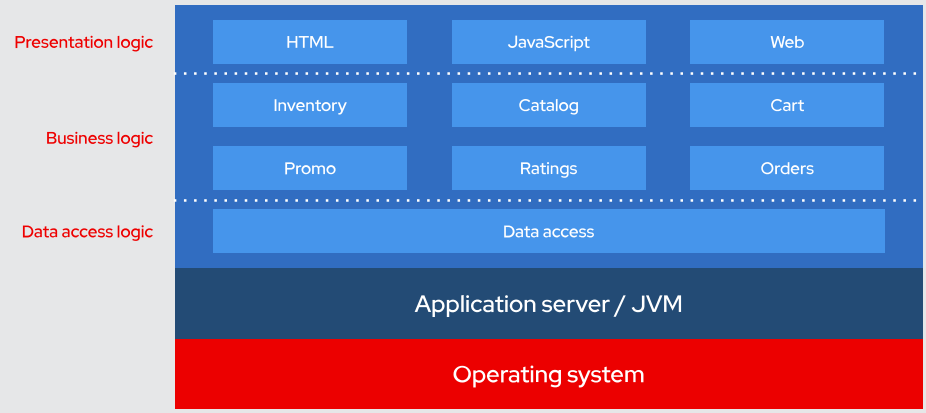
Just-in-time compiler

The just-in-time (JIT) compiler is another example of a Java feature that was important in the past for long-running applications. Over time, the JVM can learn the access patterns of an application and optimize how it serves requests to eliminate unnecessary code and maximize performance of code that is used often.

Similar to GC, the JIT compiler poses a downside in that it runs in the same process as the application and competes with the application for resources. Also similar to GC, the JIT compiler is not needed in cloud environments because applications only run for milliseconds.

The vertical stack

Multiple applications to run on one machine



The application server is another design assumption that does not fit well with a massive scale, distributed architecture. The application server was created to allow users to maximize the value of the underlying hardware. Because Java could provide high performance for multiple applications on one server, an organization could invest in one very large, very expensive application server to host all of its applications.

In theory, the application server made sense, but the downside was that applications sharing the same application server could not be isolated enough to either deliver applications continuously or run into technical issues (e.g., memory leaks) that would impact other applications, or possibly take the whole system down. As a solution to this problem—maybe more appropriately categorized as a workaround—users would deploy one application per server, completely defeating the original purpose of the application server.

In cloud environments, the application server concept is challenged due to the inherent distributed nature of the cloud computing. Applications are hosted on cloud platforms, distributed across multiple machines. This is made possible because the applications are designed to be lightweight, modular, and portable—ideal for cloud environments. In this scenario, there is no need for massive, on-premise hardware. In fact, eliminating the need for expensive on-premise hardware, like application servers, is one the most attractive benefits of cloud environments.

Designed for throughput

In traditional Java development, applications are designed in a vertical stack, which means all related components are included in the stack, all part of one monolithic application. Java was designed for maximum throughput and low latency, which requires significant resources in a vertical stack. However, this limitation no longer applies to applications in cloud environments because they can be scaled horizontally, meaning the application is distributed across multiple machines and even multiple cloud environments. In this way, the application can achieve throughput and scalability.

Why are developers moving to cloud computing?

Cloud computing offers many advantages. Here are some:

Streamlined developer experience

One aspect of cloud computing that is very attractive to developers is its ease of use for an improved developer experience. In cloud environments, a developer can get access to resources in minutes with the push of a button. In addition, the tools in cloud environments are well integrated and developers do not have to configure the network, storage, compute, or other services, which are all pre-configured and ready to go in cloud environments.

High availability

Cloud providers guarantee more than 99% uptime—a service-level agreement (SLA) that is hard to equal in a datacenter without excessive spending and substantial effort. This makes cloud services an attractive option for anyone wanting to run any type of cloud applications.

Scalability

Scalability is one of the most important reasons developers move to cloud computing, because it lets users increase or decrease resources on demand, based on changing needs. To gain the same level of scalability in a non-cloud environment would be prohibitively expensive.

Global reach

Cloud environments enable global reach, providing low-latency services all over the world. The major cloud providers offer availability zones, which are physical datacenters placed strategically in various regions worldwide. This lets cloud users extend the reach of their own applications, deploying them to customers in different parts of the world that may not have been previously practical, economical, or even possible.

Low cost

Maybe the most appealing advantage of cloud computing is the cost. Unlike datacenters and application servers, cloud computing does not require a massive upfront expenditure, and you save on the incidental costs of running the hardware—such as power, space, and maintenance. Since you pay a fixed cost, it's easier to fund your cloud initiative because you can fund it with OpEx instead of CapEx.

Cloud computing can be especially cost-effective for small companies that want to jump-start developing new applications right away. Cloud deployment is rapid and painless, and the cloud environment is easy to access and learn, consequently accelerating developer productivity and time to market, which further reduces cost.

Requirements for a modern language for cloud computing

If you decide that the advantages of cloud computing are too great to pass up, but you still want to keep using Java, it is important to understand how Java can work in a cloud environment. Java was not originally designed for cloud computing and must therefore evolve to fit this new distributed environment.

Here are the requirements that Java must meet to become a modern language for cloud computing:

Inner and outer loop development

For a traditional monolithic application, the developer will code and test locally, in an environment that replicates the datacenter. Keeping the development environment similar to the production environment is helpful because when the application finally moves into production, it has already been tested and proven in a production-like environment.

Cloud-native development has changed this process because it is impossible to replicate Kubernetes and the cloud environment onto your local development environment as was done previously. Consequently, the development environment is now remote, in a cloud environment that is similar to the production environment. This means a modern language for the cloud environment must be able to support remote development.

Integration with cloud-provider services

A modern language and framework for cloud computing must integrate with services offered by the cloud provider so developers can use cloud services such as AI, data lakes, Apache Kafka, and API management.

Seamless experience across all hyperscalers

Many organizations that move to cloud environments use multiple cloud providers to balance the risk and avoid vendor lock-in. One of the most important requirements for developers working in this environment is a consistent experience across any combination of cloud providers—with a familiar platform and tooling. That is why a modern language for cloud computing must be cloud-environment-agnostic.

Cloud architecture patterns

Cloud computing provides several innovative types of architectures that are changing the way applications are developed. These architecture patterns are distributed and do not rely on a vertical stack. They require developers to build applications in a modular way, composed of smaller components that can communicate and scale independently across multiple clouds.

Here are three major cloud architecture patterns that represent software development in cloud computing:

Event-driven architecture

Event-driven architecture (EDA) is a software development method for building applications that asynchronously communicate or integrate with other applications and systems via events.

An event can be any occurrence or change in state that is identified by the application. An application designated as a "producer" detects the event and sends out relevant data in the form of a message. Multiple "consumers" can receive the same message and use or process the associated data in their own way to accomplish the specific job for which the application is designed. EDA architecture can be used for any type of communication or data transfer.

Asynchronous event-based communication differs from the more traditional synchronous communication method in which two applications make a direct connection, most commonly via application programming interfaces (APIs). Conversely, asynchronous communication is event-driven, allowing multiple applications to communicate simultaneously and rapidly in real time.

EDA requires minimal coupling between the services while they can still communicate with each other,

which makes EDA optimal for modern, distributed applications built in cloud environments.

The advantages of EDA include:

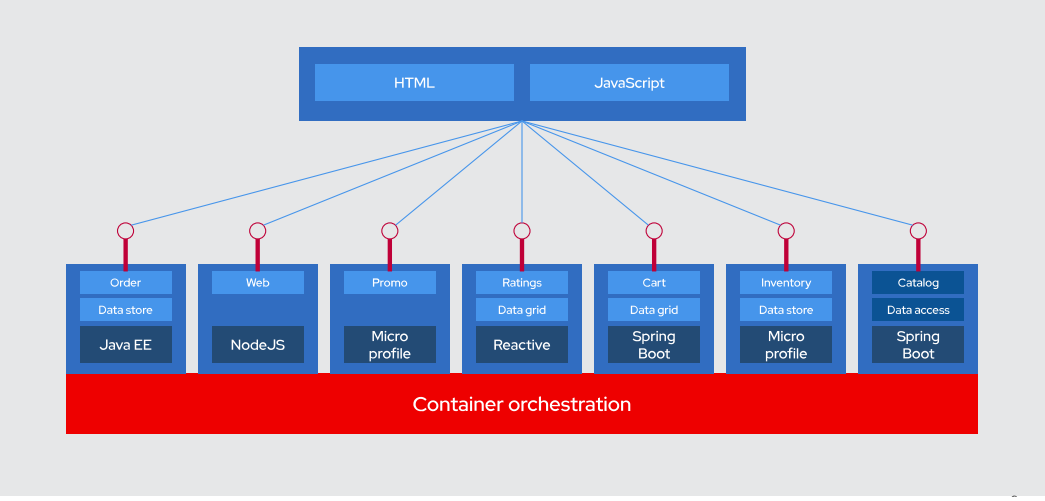
- Real-time communication.
- Low latency.
- Scalability.
- High throughput.
- Reliability.

Conversely, traditional synchronous Java applications are not designed for a cloud environment and face challenges in a distributed environment, such as communication latency, system degradation, and unpredictable failures.

EDA is an ideal architecture for enterprise applications that derive benefits from scalable and reliable real-time communication, such as stream processing, data integration, and website activity tracking. In addition, EDA is often employed where humans generate the interaction, like the addition of a product to a shopping cart on an e-commerce site or a retail kiosk.

Microservices architecture and service mesh

Microservices architecture is a software development method that has revolutionized the way applications are built and deployed. Unlike the traditional monolithic approach to development, which involves building all the application's components into a single deployment unit, the microservices approach breaks down the application into independent components that represent core functions or services. These services are loosely coupled, which means that while they are independent, they can still communicate with each other and work together as an application.



Microservices represent a key step in optimizing application development for a cloud-native model. Because an application is made up of independent services, a microservices architecture is perfect for cloud computing. The application can be distributed across multiple cloud environments and even on-premise hardware.

The main advantages of microservices are speeding up the development life cycle and reducing time to market. Because an application is made up of a collection of separate services, these microservices can all be built, tested, and deployed independently by multiple developers. This method minimizes delays typically caused when developers have to wait for an entire application to recompile and redeploy every time any type of change is made. When using microservices, the change can be made to a component of the application without impacting the rest of the application. In this way, updates and fixes to applications can be implemented much more easily and quickly.

In addition to faster development, microservices provide these other advantages:

- **Quality:** Because microservices are small, it is easier for developers to find and fix problems, thereby producing a higher-quality application.
- **Scalability:** Applications can scale quickly and economically because the microservices can be deployed across multiple cloud environments, scaling to meet demand.
- **Resiliency:** As independent services, microservices do not impact each other, so the whole application does not go down when there is a failure within a single service.
- **Innovation:** Microservices give developers freedom to be more innovative because they can make changes and experiment without worrying about impacting the application and slowing time to market.

For a microservices architecture to work as a functional cloud application, services must constantly request data from each other through messaging. Building a service mesh layer simplifies interservice communication.

A service mesh is a transparent, dedicated infrastructure layer that resides outside of an application, designed to control how different microservices that make up an application share data with one another. The purpose of the service mesh is to optimize service-to-service communications and minimize potential points of communication failure.

A service mesh removes the logic governing service-to-service communication out of individual services and abstracts it to a layer of infrastructure. This is accomplished by providing insight into—and control of—the networked services within the mesh through the use of a sidecar proxy that intercepts network communication between microservices. A sidecar proxy sits alongside a microservice and routes requests to other proxies. Together, these sidecars form a mesh network.

A service mesh can be very helpful in cloud environments because cloud-native applications are built using microservices, and as more microservices are added to applications, managing communication in a cloud environment can become complex and challenging. The advantage of a service mesh is providing connectivity between microservices and making communication flexible, reliable, fast, and manageable.

Without a service mesh, each microservice must be coded with logic to enable service-to-service communication, which takes developer time and focus. A service mesh streamlines the development process by eliminating the need for this extra coding.

A service mesh also enables developers to introduce added capabilities to microservices, such as

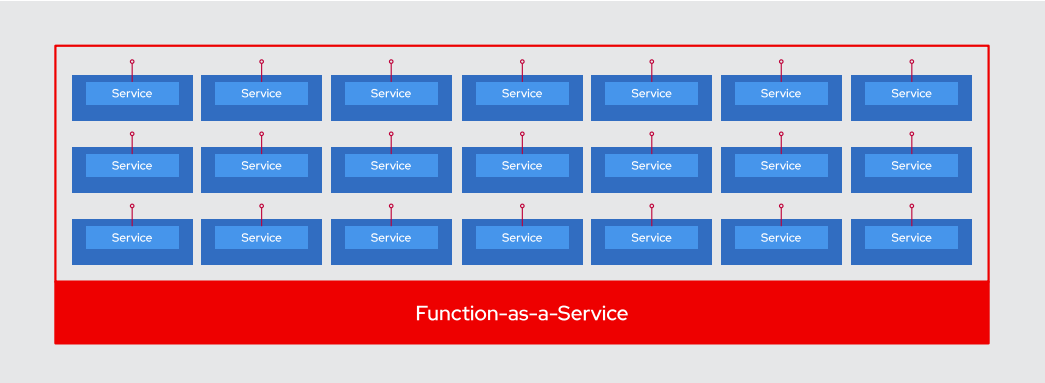
routing control, fault tolerance, and security while making no changes to the microservice components themselves.

In addition, a service mesh provides observability, monitoring, and testing to ensure communication performance and availability. A service mesh makes it easier to troubleshoot communication failures, because the source of the problem is not hidden within the microservice, but rather, in a visible infrastructure layer alongside the services. Ultimately, this makes applications more robust and less vulnerable to downtime.

A service mesh also has a self-learning capability to further improve communication. The service mesh captures performance metrics on service-to-service communication and uses that data to automatically improve efficiency and reliability of service requests.

Serverless architecture

Serverless architecture is a cloud-native development model that allows developers to build and run applications without having to manage the underlying infrastructure. A cloud provider handles the routine work of provisioning, maintaining, and scaling servers so developers can focus on coding.



Serverless applications are deployed in containers that automatically launch when called. The application can be triggered by a variety of sources, such as events from other applications, cloud services, Software-as-a-Service (SaaS) systems, and other services. Once deployed, serverless applications automatically scale up or down according to demand. For this reason, serverless applications only consume resources when they are needed.

Serverless differs from a standard Infrastructure-as-a-Service (IaaS) cloud computing model. In the traditional scenario, the user pays the cloud provider for an always-on server to run applications. The cloud infrastructure necessary to run an application is active even when the application is not being used.

With serverless architecture, however, applications are launched only as needed. When the application runs, the cloud provider allocates resources. The user only pays for the resources while the application is running.

The typical serverless model is Function-as-a-Service (FaaS), an event-driven design model allowing developers to write logic that is deployed in containers managed by a cloud provider, then executed on demand. The application is triggered by an event, and run automatically when needed.

Serverless is a good fit for applications that experience infrequent, unpredictable surges in demand.

Serverless is also useful for use cases that involve incoming data streams, chat bots, scheduled tasks, or business process automation.

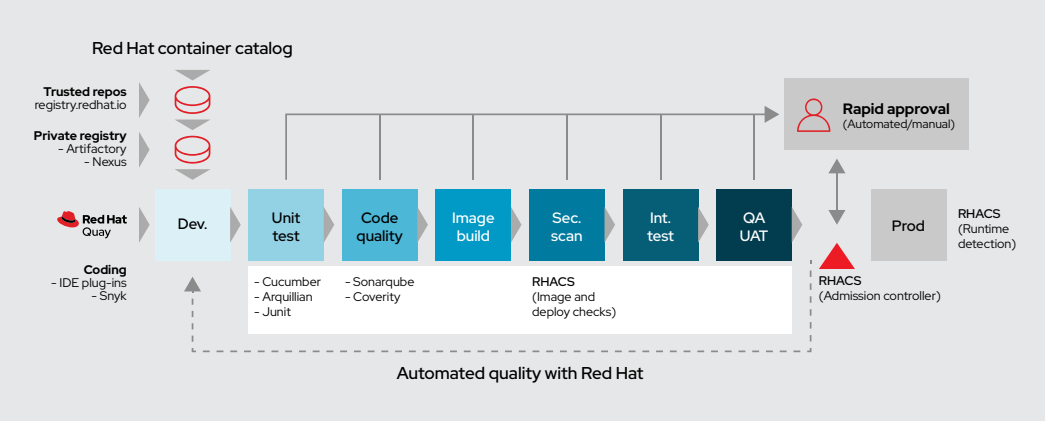
In addition to the obvious benefit of reduced costs, serverless lets developers scale and distribute workloads across multiple cloud environments. The serverless model is so flexible that applications can be scaled from zero to extremely high resource, used, and then scaled back to zero on a regular basis.

Also, serverless frees developers from routine, time-consuming tasks associated with scaling and server provisioning, so they can spend more time innovating applications and features with business value. With serverless, tasks such as managing the operating system, file system, and security patches; load balancing; capacity management; scaling; logging; and monitoring are all offloaded to a cloud provider.

Traditional Java frameworks are typically too heavyweight and slow for serverless deployment to cloud environments, especially deployment on Kubernetes. To work in a cloud environment, a Java framework must evolve to provide faster startup time, lower memory consumption, and smaller application size.

Focus on DevOps/DevSecOps

DevOps is an approach to IT culture that brings development teams and IT operations teams together into a more collaborative, end-to-end development process. In the DevOps approach, both development and IT Ops are responsible for developing and deploying high-quality applications. This often means bringing IT Ops requirements into the development cycle at an earlier stage to ensure the application is built with production requirements in mind. When done correctly, DevOps can streamline the application development life cycle and bring better applications to market faster.



DevSecOps is a similar approach that also brings security teams into the process earlier than normal to work together with development and IT Ops to ensure applications are secure as well.

The key to DevOps and DevSecOps is collaboration through sharing visibility, feedback, lessons learned, and other insights. DevOps and DevSecOps approaches also focus on automating the development cycle as much as possible, to improve time to market. All of the objectives require the right platform and tools to support a collaborative and automated development environment.

DevOps goes hand in hand with cloud-native development because both initiatives work toward the same goal: faster development of quality applications. A modern language for cloud computing must inherently offer capabilities to support DevOps, including:

CI/CD: Continuous integration, continuous delivery, and continuous deployment are DevOps concepts that support cloud-native development. Continuous integration means changes made to an application by different developers are merged in a shared repository to ensure that all developers working on the application have access to the same version of the application. Continuous delivery or continuous deployment means changes to an application are automatically tested and deployed to streamline the development process. A modern language for cloud computing must support the CI/CD pipeline to gain the benefits of accelerated development and improved quality.

Automation: In cloud-native development, automation is very important because of the complexity of distributed applications in the cloud environment. Multiple applications made up of multiple microservices spread across multiple cloud environments can be a challenge to develop and manage. Automation of processes like integration, testing, and deployment can alleviate some of these challenges, and significantly accelerate the development process while improving quality due to reduced human error.

Bringing Java into cloud computing with Quarkus

Quarkus is a Kubernetes-native Java stack built on proven Java libraries and standards and designed for containers and cloud deployments. Quarkus lets organizations maximize years of investment in Java by translating the language for cloud computing and Kubernetes. In this way, Quarkus provides a pathway for developers to continue using their existing Java knowledge and experience, as well as the same Java frameworks they have used in the past.

Quarkus was designed to address Java's limitations with regard to cloud-native application architectures like containers, microservices, and serverless. Quarkus is the ideal Java framework for hybrid cloud computing because it supports both traditional and cloud-native architectures.

Java developers are able to use Quarkus to build apps that have a faster startup time and take up less memory than traditional Java-based microservices frameworks. This translates into lower costs because it takes less memory and CPU to run the same application. It also translates into higher productivity because developers are not waiting around for massive applications to rebuild and redeploy to test their changes. With Quarkus, a developer can make a change to the code and see it in the application immediately.

Quarkus works out of the box with popular Java standards, frameworks, and libraries, alleviating the need to learn new APIs or switch to another programming language altogether. So the developer can choose the Java framework they want to use with Quarkus, but then Quarkus adds developer productivity and fast performance on top.

These are the capabilities, features, and tools that Quarkus adds on top of Java:

Less code

When using Quarkus, developers are able to write less code and still build more functionality compared to traditional Java. This helps the sustainability of the application. With a super lightweight, super fast application built using Quarkus, you can deploy hundreds of copies of the application and achieve extreme throughput without adding footprint.

High performance

Quarkus does not simply translate Java to the cloud—Quarkus takes Java to the next level. Over the last decade, there have been attempts to make Java faster, but all those attempts focused on the Java VM, the underlying Java technology, and not the frameworks that ride on top. Quarkus optimizes the frame-

works themselves to achieve blazingly fast performance.

Users can deploy more applications in Quarkus with the same resources, compared to traditional Java. The high throughput delivered by Quarkus comes from the ability to have very dense deployments of multiple copies of Quarkus distributed across the network.

Quarkus was built on a container-first philosophy, meaning it is optimized for lower memory usage and faster startup times. Quarkus builds applications to consume 1/10 the memory when compared to traditional Java and has a faster startup time—as much as 300 times faster.

Live coding

Quarkus enables fast iteration during development with "live coding." Code changes are automatically and immediately reflected in the running application.

The traditional Java workflow requires developers to recompile and redeploy an application each time a change is made, taking up to a minute or more. This adds up to significant delays for Java developers. Live coding helps increase developer productivity by letting them make changes on the fly, and simply refresh the browser, without having to recompile and redeploy the entire application every time. Quarkus typically implements those changes in under one second.

Continuous testing

Continuous testing, available in Quarkus at the press of a key, lets developers pursue test-driven development.

In the traditional Java development life cycle, the developer will write code, write a test, run the test, see if the test passes or fails, and then make changes. In Quarkus with continuous testing, the developer can run unit tests continuously in real time, concurrently while writing code. The tests run automatically in the background, providing constant feedback. Any time you make one wrong keystroke and cause a test to fail, you know about it immediately. This capability substantially accelerates the development cycle.

Dev services

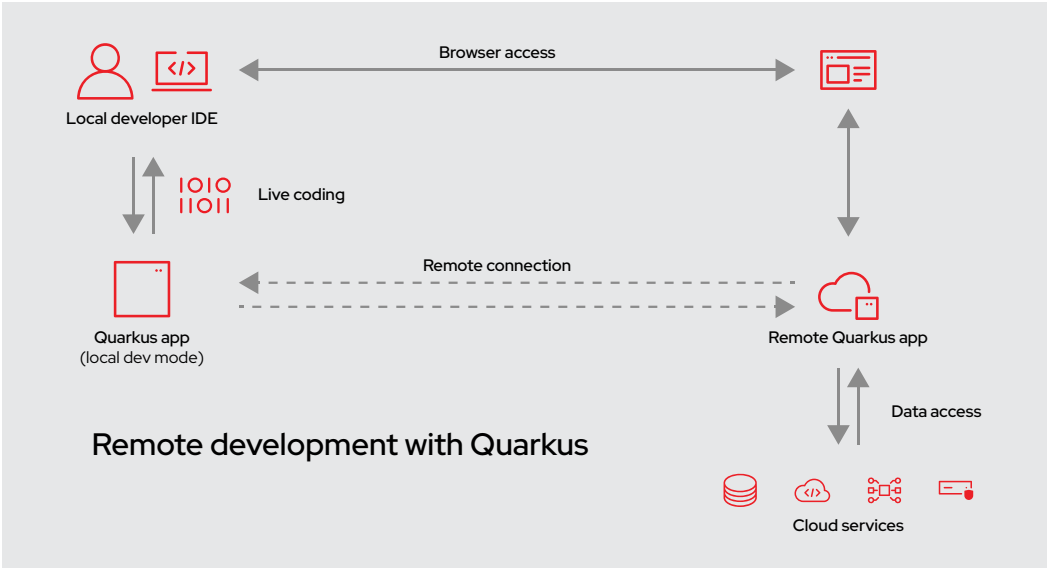
Quarkus dev services let developers easily test application dependencies.

Every application has dependencies on other applications and services. During traditional Java development, the developer would have to spin up a copy of any connected service to fully test the application. However, some services—like Apache Kafka, message brokers, and identity management systems—can be very challenging to replicate. Quarkus solves this problem with dev services, which automatically provide services needed to test an application.

For example, if your application needs a database, Quarkus identifies this need, knows the correct database, spins up the database, and connects your application—all automatically.

Remote Dev

The ability to do remote development in a cloud-native Java runtime simplifies the development workflow—from writing code to building, running, debugging, and deploying microservices at speed.



Quarkus remote dev lets the developer run applications in a remote container environment while still gaining access via their local laptop. Changes made on the local development machine are automatically pushed to the running remote Quarkus application in real time.

The advantage of remote dev is that the remote environment is closer to the application's production environment making tests more accurate. A related advantage of remote dev is access to services in the cloud environment that may not be available or easy to recreate on a developer's local machine. Overall, remote dev can give the developer confidence that the application will run in production, and drastically reduce the time needed to develop and test changes.

Cloud and Kubernetes-native development on Red Hat OpenShift

Historically, it has been difficult to deploy applications to Kubernetes. However, Quarkus has innate knowledge of Kubernetes and Red Hat® OpenShift®, Red Hat's underlying Kubernetes-based platform. Quarkus automates the process of deploying your application on Kubernetes and Red Hat OpenShift, handling all the configuring and script writing giving you a Kubernetes-native development experience.

Simply adding the Quarkus Kubernetes extension or the Quarkus OpenShift extension will automatically generate Kubernetes or OpenShift resources based on the project.

Integration with cloud services

Quarkus also has innate knowledge of the many services offered by the major cloud providers allowing developers to easily integrate applications with these services.

For example, Quarkus Funqy is a portable Java API used to write functions deployable to Function-as-a-Service (FaaS) environments like AWS Lambda, Azure Functions, Knative, and Knative Events.

Another example, Quarkus lets developers use AWS services such as S3, SNS, and Alexa, as well as other cloud provider services.

Integration with traditional Java frameworks

Quarkus allows developers to continue to use the Java frameworks they know and love. Quarkus is designed to work with popular Java standards, frameworks, and libraries like Eclipse MicroProfile and Spring, as well as Apache Kafka, RESTEasy (JAX-RS), Hibernate ORM (JPA), Infinispan, Camel, and many more.

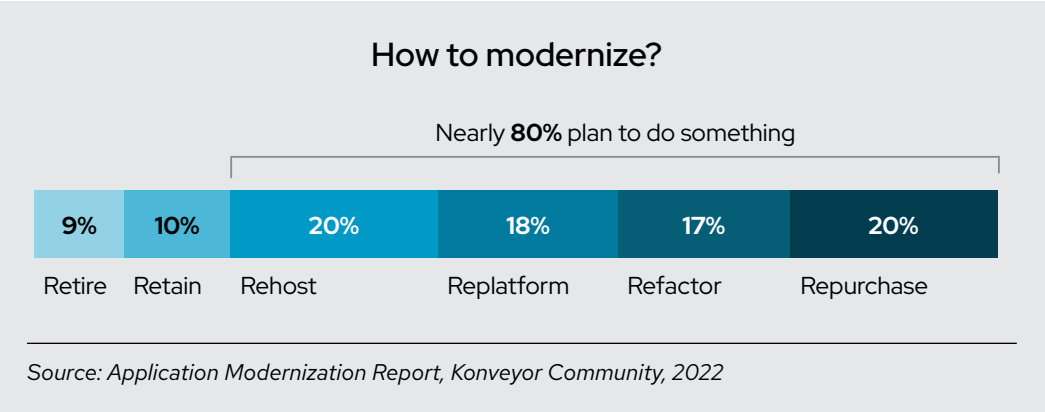
Integration with CI/CD, observability, tracing, telemetry

When you move from traditional Java to a distributed architecture like microservices, your application becomes more complex. Using distributed services and EDA, challenges in observing, tracing, and debugging become exponentially greater. Quarkus minimizes the pain by providing out-of-the-box support for observability, tracing, telemetry, and CI/CD systems.

Your pathway to the cloud

If you are thinking about modernizing your Java applications, moving away from monolithic architectures, and taking true advantage of the cloud deployment model, there are three main paths to Java application modernization: rehost, replatform, and refactor.

Keep in mind that all your applications do not need to follow the same modernization path. You can choose the path that best fits the characteristics of each application, as well as your organization's current and expected needs.



Source: Application Modernization Report, Konveyor Community, 2022

Path 1: Rehost

The rehost path, also known as "lift and shift," means to deploy your existing application as is within a virtual machine (VM).

The rehost path involves lifting and shifting Java applications running on traditional application servers into VMs running on a hybrid cloud platform. Monolithic applications remain unchanged on your application server and retain all existing integrations and dependencies. External data and integrations can still be kept on your legacy platforms.

Rehosting generally takes a short amount of time and results in low migration costs, but it delivers fewer benefits than the other modernization paths. Keeping this in mind, rehosting can still help you move applications to a consistent platform, and it can prepare you for future cloud-native operations.

Some traditional application servers may not work in a VM. In this case, you must redeploy your applications in a modern runtime environment before moving them to a VM. If you need to change runtime environments, consider replatforming (path 2) your applications and deploying them in containers to optimize your modernization efforts.

Path 2: Replatform

The replatform path means to deploy your application in a container on a Kubernetes-based cloud platform.

Replatforming involves lifting, modifying, and shifting Java applications to modern runtime environments running in containers on a hybrid cloud platform. Basic Java applications require few changes to benefit from a containerized Java runtime like OpenJDK.

Enterprise applications are migrated to modern runtime environments—like Red Hat JBoss Enterprise Application Platform, IBM WebSphere Liberty, or Red Hat JBoss Web Server—prior to deploying them in containers. This path usually takes longer than rehosting, but it delivers more benefits. Unifying your applications on a single hybrid cloud platform streamlines operations and allows you to deliver self-service capabilities. Your replatformed applications can also take advantage of all of the native capabilities of your hybrid cloud platform.

Path 3: Refactor

The refactor path means to rebuild your application as microservices, integrate new technologies, and deploy on a cloud platform.

Refactoring involves redeveloping Java application services as microservices deployed within a service mesh on a hybrid cloud platform. Services can be rebuilt over time to gradually move functionality from your old application architecture to your new one.

During the process, you can also upgrade underlying technologies and add in new cloud-native capabilities like artificial intelligence and machine learning (AI/ML), analytics, autoscaling, serverless functions, and event-driven architecture.

Refactoring takes the most time, but it also provides the greatest advantage. Refactoring delivers all of the benefits of rehosting and replatforming while allowing you to take advantage of innovative new technologies to increase business agility and value.

Customer testimonials

Here are just a few examples of Red Hat customers who successfully moved Java into cloud environments with Quarkus:

The Asiakastieto Group

The Asiakastieto Group is a leading provider of innovative digital business and consumer information services in the Nordic region. To support a shift to open banking, comply with new data privacy and security requirements, and help solve high levels of debt in the Nordics, the company built a credit assessment solution. Using Red Hat OpenShift, Red Hat Integration, and Quarkus, Asiakastieto developed their Account Insight application to reduce personal debt and payment defaulting to a more accurate assessment of an individual's repayment ability. The company also migrated from Thorntail to Quarkus to optimize its hardware resources by improving microservice start-up times, memory use, and server density.

Bankdata

An IT service provider for several large Danish banks, Bankdata builds, implements, and runs high-quality IT solutions. To find the best Java framework, Bankdata tested the performance and efficiency of their Spring Boot Java framework against Quarkus. The results showed that a Quarkus-native version of the test application provided faster boot-up times (less than one second compared to 3 minutes for Spring), 57% less memory use per call processed, and lower CPU use. Bankdata found that migrating to Quarkus could reduce testing time so the company could bring innovative new services to market quicker.

Lufthansa Technik

Lufthansa Technik runs a digital platform, called AVIATAR, that helps airlines avoid delays and cancellations by better organizing and scheduling of maintenance. To address fast growth and increasing demand from customers, the company decided to move to a microservices architecture based on Microsoft Azure Red Hat OpenShift. The AVIATAR team deployed Quarkus to help reduce cloud resource consumption. Lufthansa Technik found that the team could run 3 times denser deployments without sacrificing availability and response times of services. In addition, Quarkus helped accelerate the development life cycle. For example, a two-person team developed a new microservice, called the "Customer Configuration" service, in a single 3-week sprint.

Learn more

For complementary information on this topic, visit Why develop Java apps with Quarkus on Red Hat Openshift.

North America	Europe, Middle East, and Africa	Asia Pacific	Latin America
1 888 REDHAT1 www.redhat.com	00800 7334 2835 europe@redhat.com	+65 6490 4200 apac@redhat.com	+54 11 4329 7300 info-latam@redhat.com

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

redhat.com
#F31646_0622