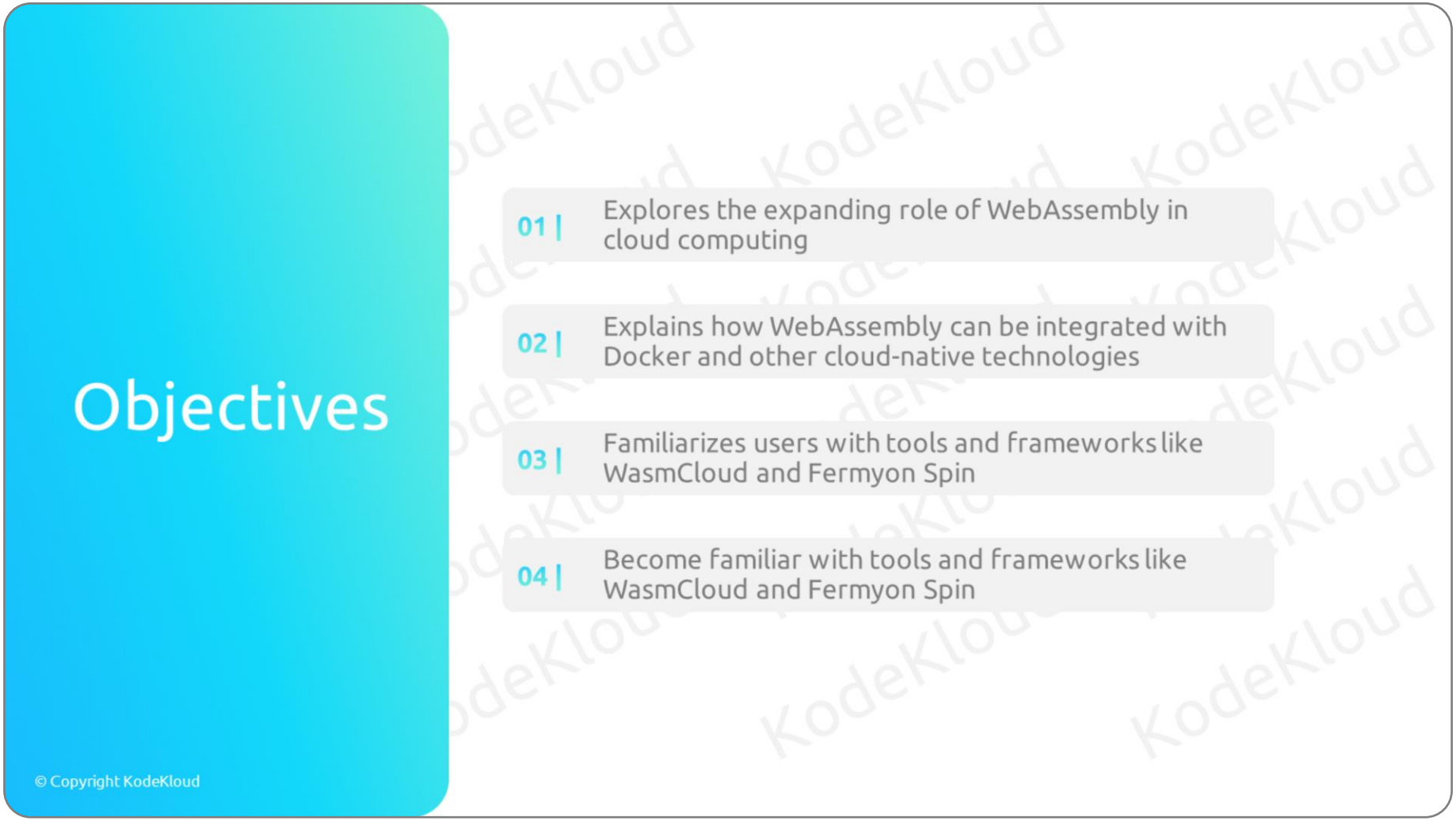
Follow us on https://kodekloud.com/ to learn more about us.

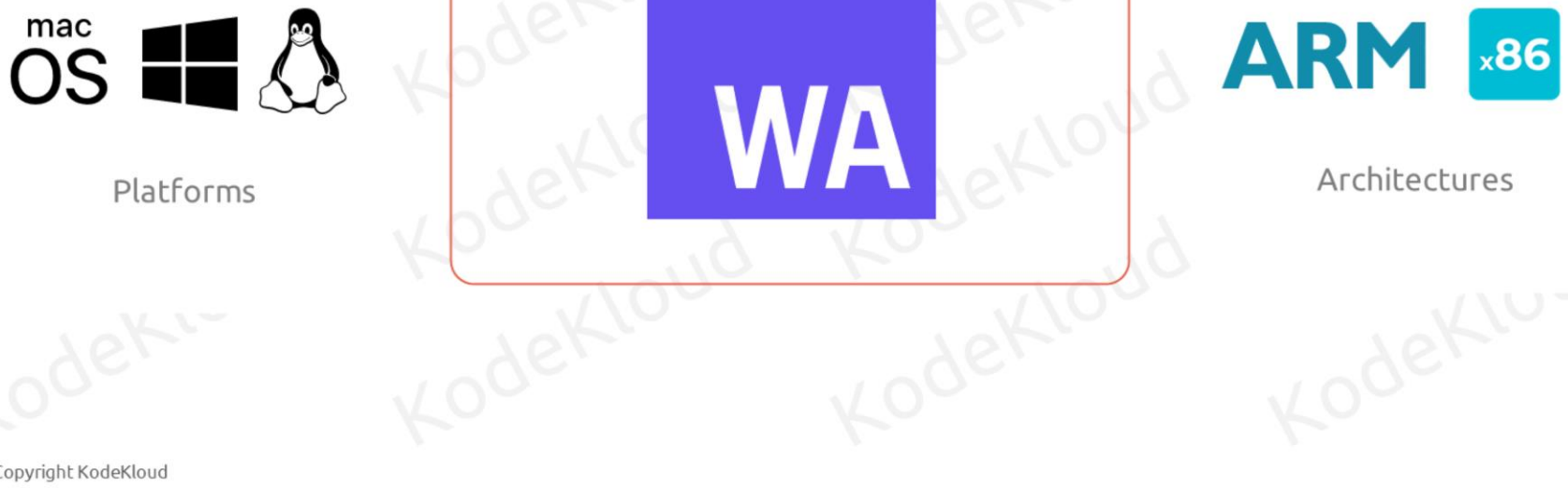
WebAssembly in Cloud – Future



Learners will explore the expanding role of WebAssembly in cloud computing. They will understand how WebAssembly can be integrated with Docker and other cloud-native technologies, and become familiar with tools and frameworks like WasmCloud and Fermyon Spin, preparing them for future developments in this area.

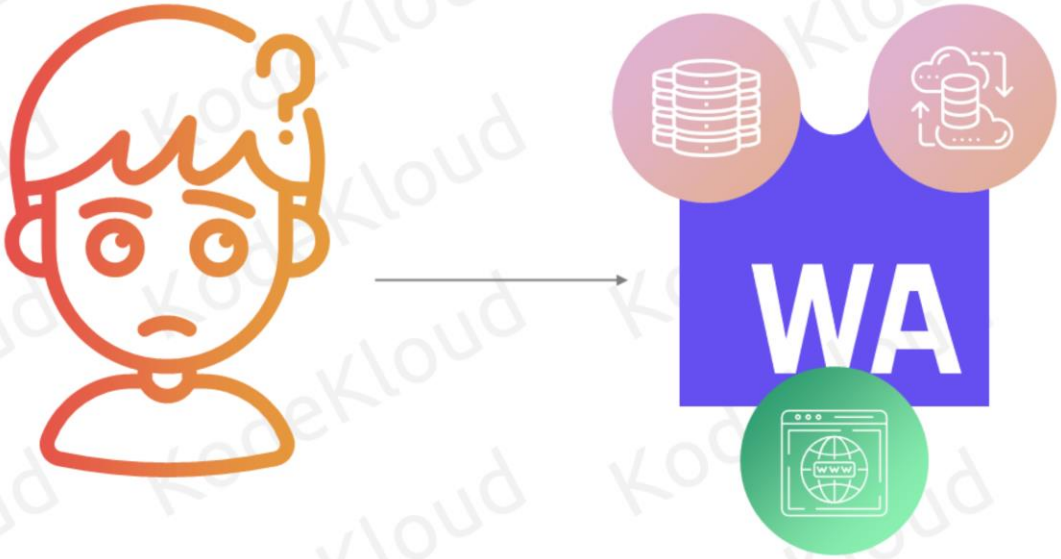
WASM Beyond the Broswer

Background



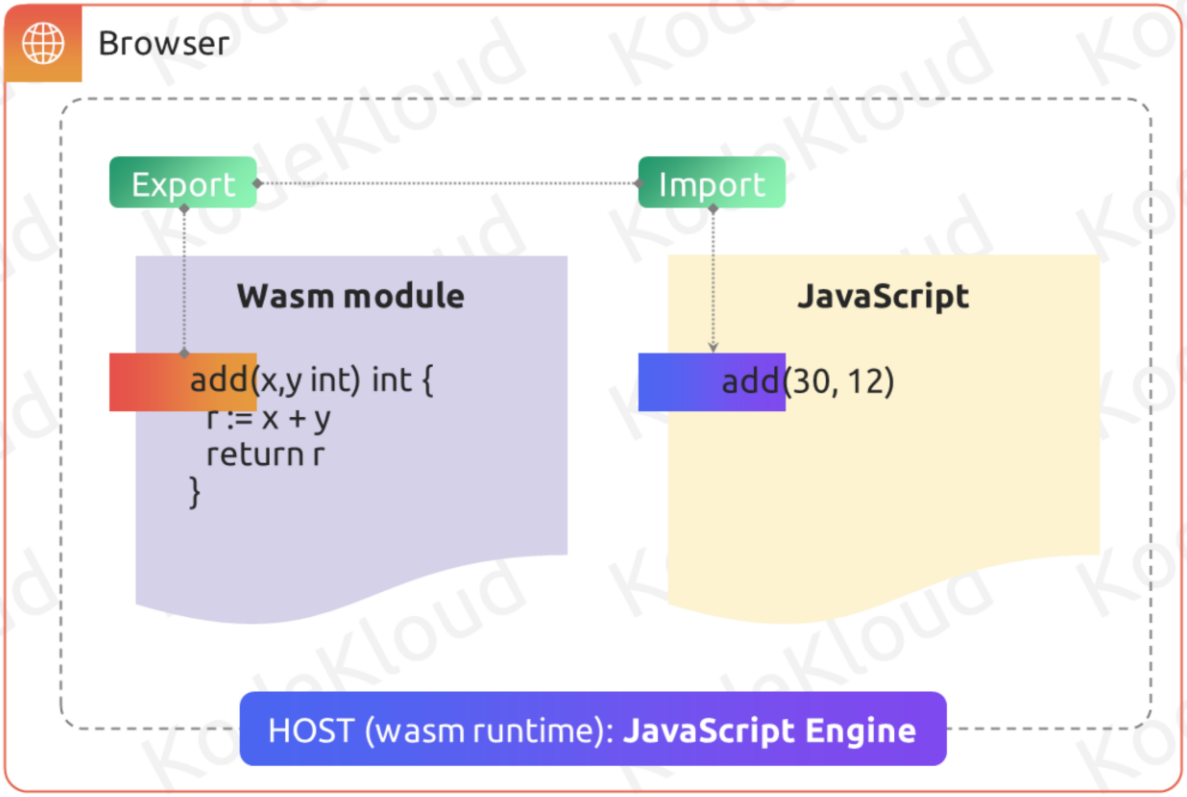
So, we all know WASM is super fast, almost like native code, works anywhere no matter the platform, and is pretty safe, thanks to its sandboxing.

Background



With all these cool features, it got some of us thinking: "Why limit WASM to just browsers? Can't we tap into its awesomeness for servers or the cloud?"

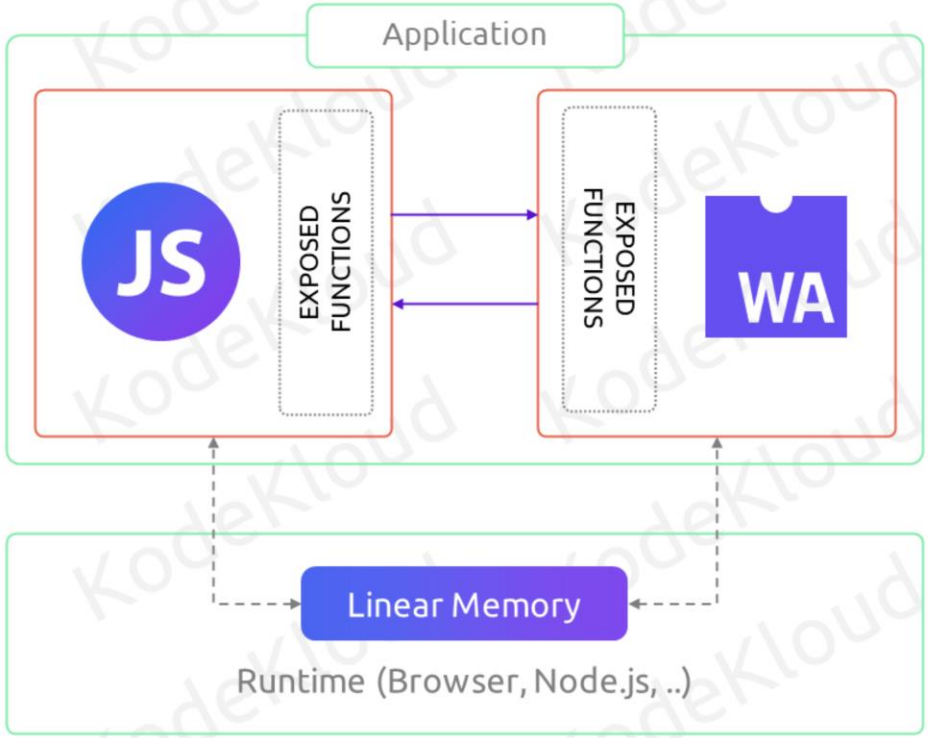
Background



Well, let's dive right in and see how the tech world responded to that thought. The tech community didn't just sit around; they took WASM for a spin in various environments:

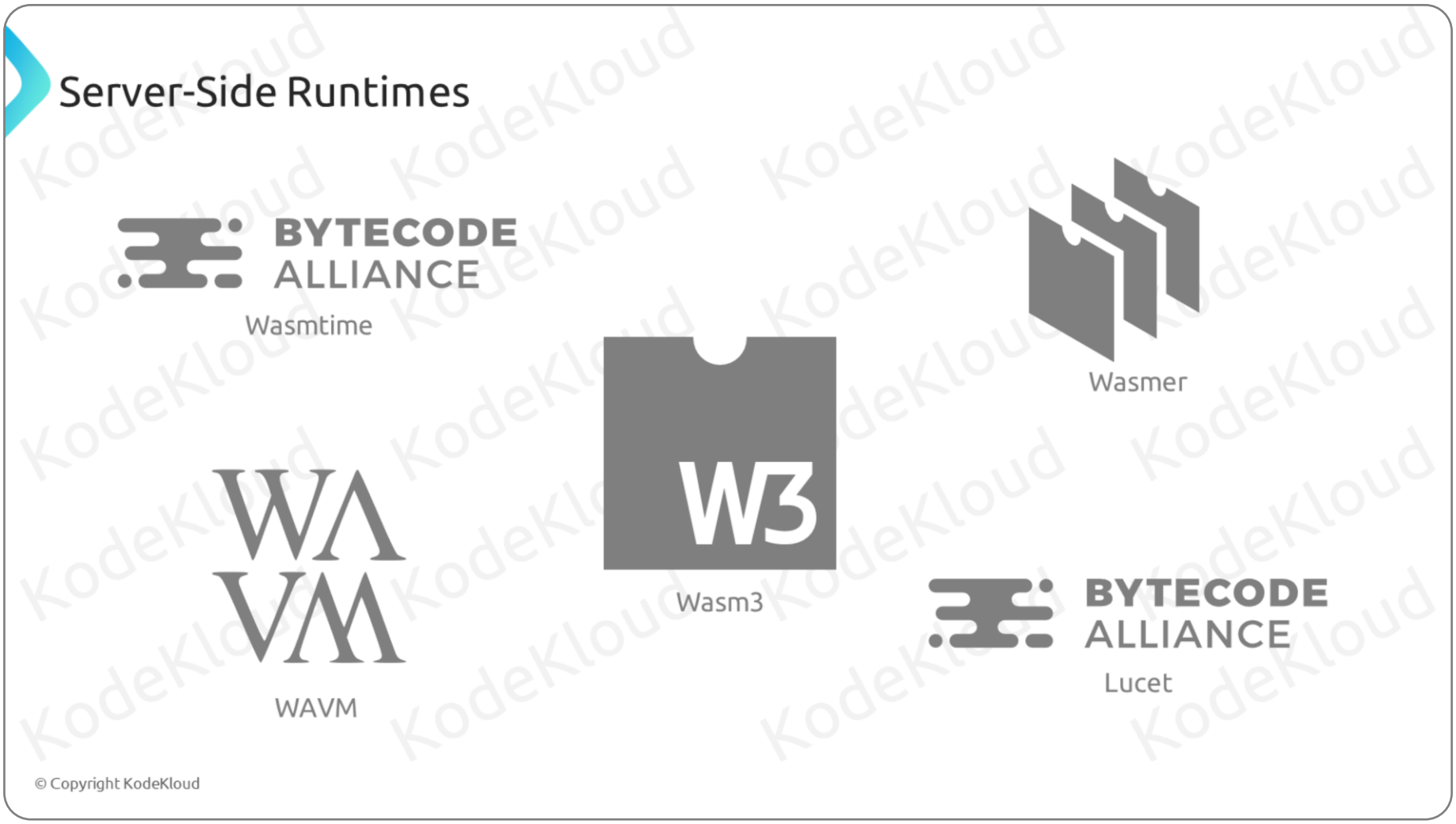
When you run WASM in a browser, it's not just floating around aimlessly. It's executed inside the JavaScript engine's VM, specifically in a dedicated environment called the WASM runtime. This runtime ensures that WASM runs efficiently and securely.

Background



One of the beauties of this setup is how WASM and JavaScript coexist. They can communicate, share tasks, and work together in harmony, all thanks to this runtime environment.

If the WASM runtime in browsers is so effective, it begs the question: Can we create a similar environment outside the browser? Imagine the possibilities if we could harness this runtime magic on servers, in the cloud, or at the edge!



First stop, Server-side Runtimes:

Tools like Wasmtime, Wasmer, Wasm3, WAVM, and Lucet are stepping up to the challenge. They're essentially creating a WASM runtime for servers, allowing WASM to run in environments beyond the browser.

Server-Side Runtimes

Server-Side Runtimes

With a dedicated runtime on servers, applications can process data faster, handle more requests, and deliver a smoother user experience.

Now, if you're wondering what these "WASM runtimes" like Wasmtime and Wasmer really are, hang tight! We'll be diving deep into the world of WASM runtimes in an upcoming lesson.

WASM in Action – Server Side

```c
#include <stdio.h>

int main() {
  printf("Hello, WASM in Server Side!\n");
  return 0;
}
```


Let's take a moment to see WASM in action on the server side.

Begin by creating a file named hello.c. In this file, we'll write a basic program to display our message:

//Code//

WASM in Action

As you're already familiar with creating WebAssembly modules using Emscripten, let's proceed with that knowledge.

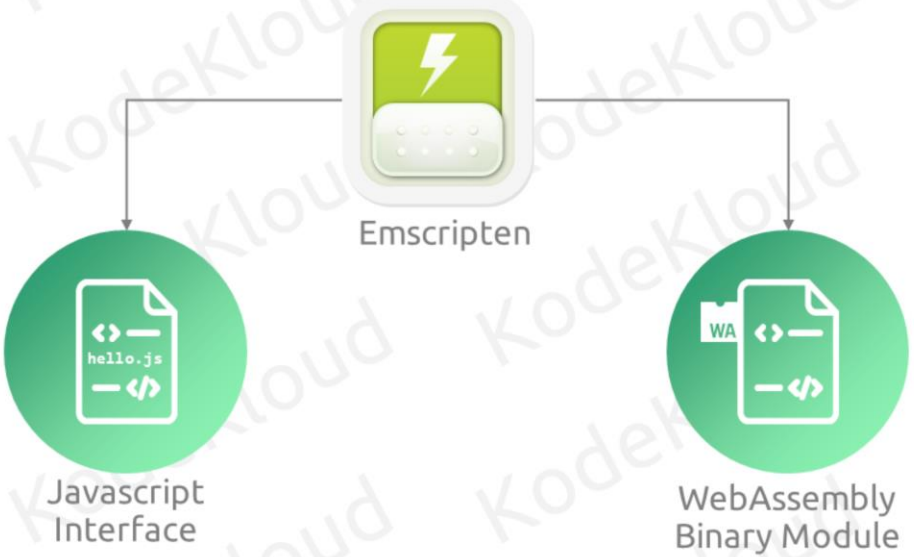
WASM in Action

```
emcc hello.c -o hello.js
```

Transform our C program into WebAssembly by running:

//Code//

WASM in Action



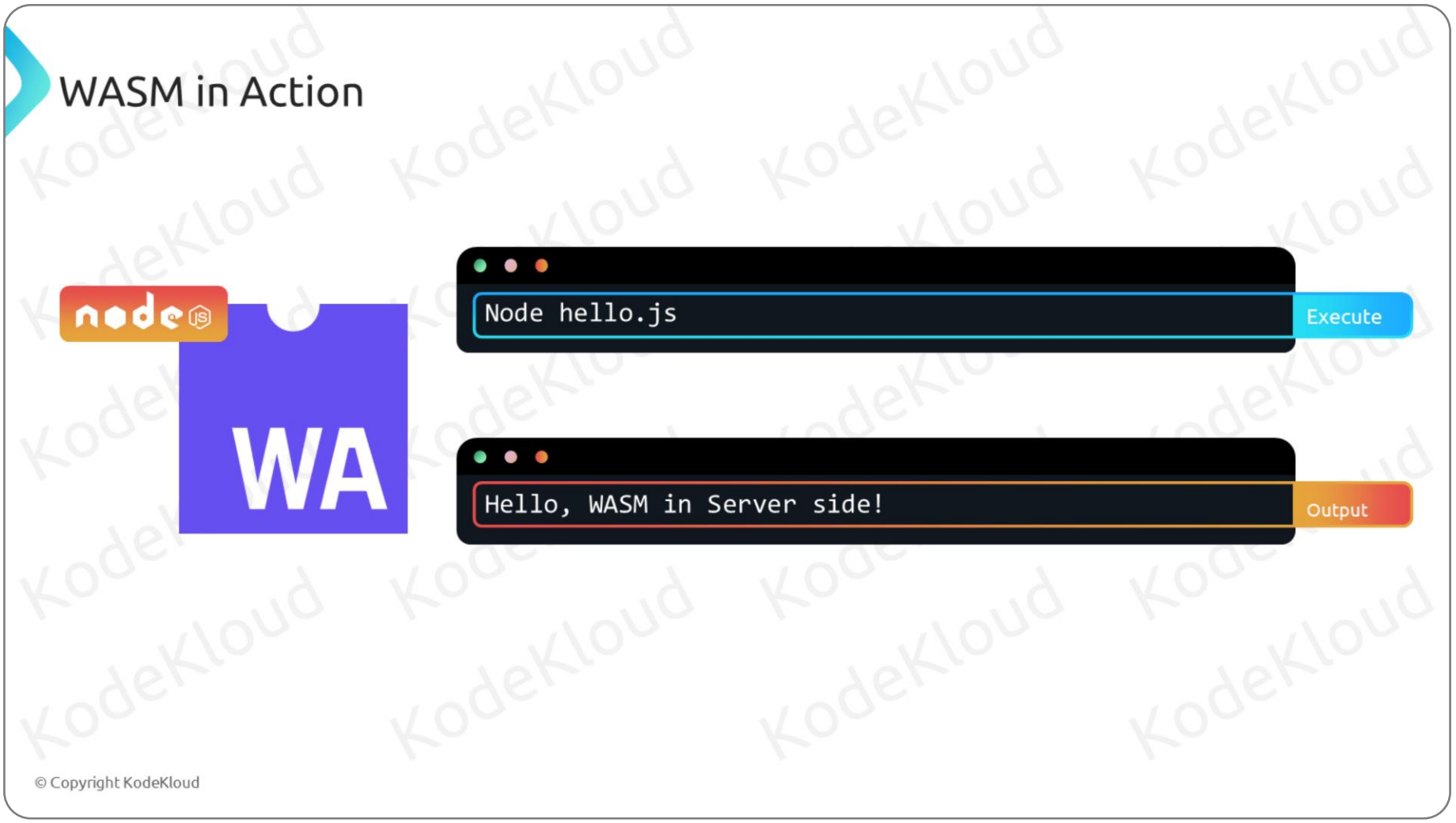
Upon successful compilation, Emscripten will produce two essential files: hello.js, our JavaScript interface, and hello.wasm, the WebAssembly binary module.

WASM in Action

In our previous lessons, we executed the hello.js file in the browser using an HTML glue. However, this time, we're venturing into a different territory: the server side with Node.js.

WASM in Action

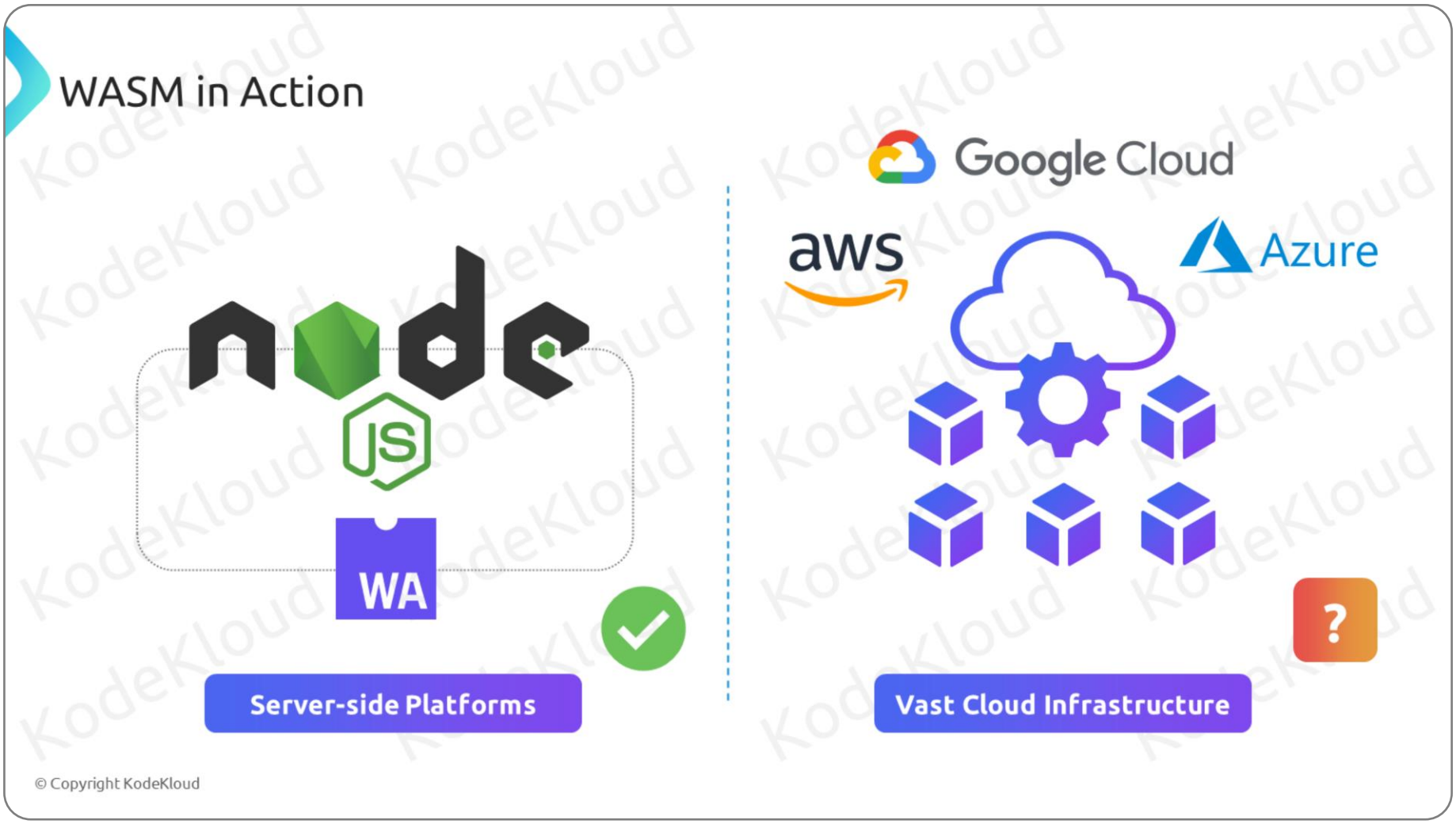
Node.js is a server-side platform built on Chrome's V8 JavaScript engine. It lets you run JavaScript on the server, not just in web browsers. This makes Node.js a great choice for demonstrating how WebAssembly can be used outside of web browsers.



With our WebAssembly module at the ready, it's time to tap into Node.js. Execute the program with:

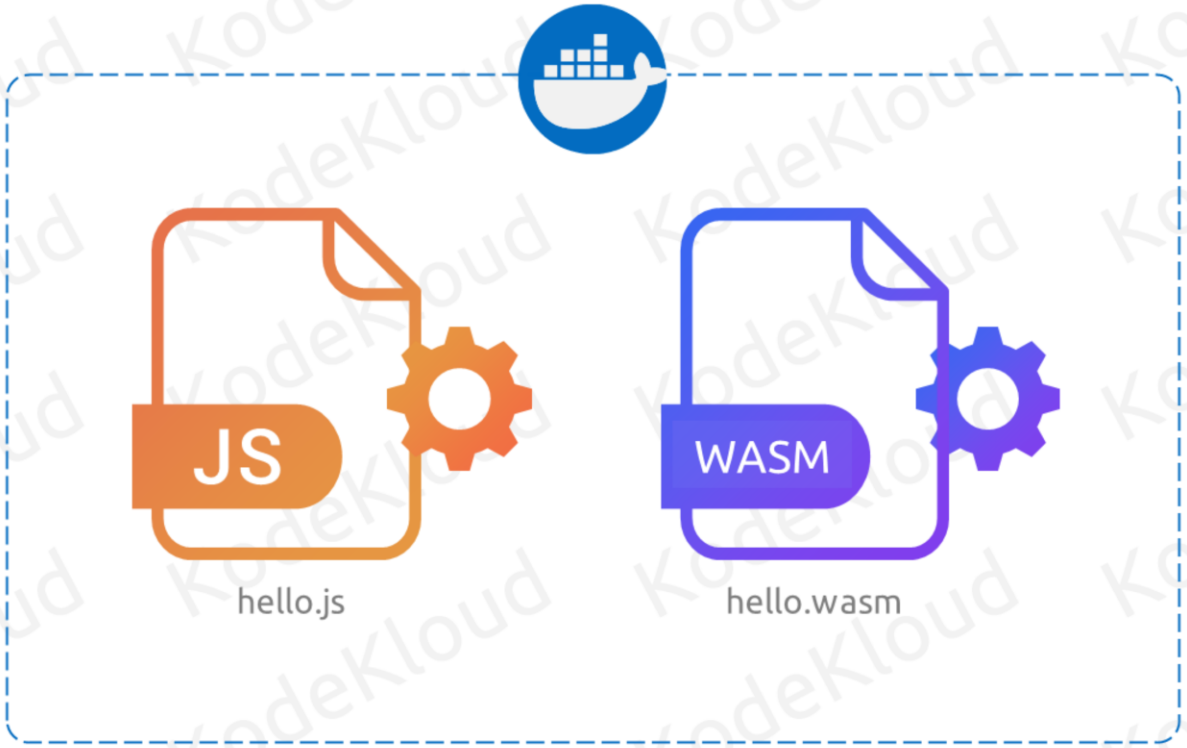
//code//

Upon running, you should see the Hello, WASM in Server side! message, signifying our C program's successful execution in a Node.js environment through WebAssembly.



Now, you might wonder, if WebAssembly can run efficiently on a server-side platform like Node.js, what's stopping us from taking it further? What about the vast cloud infrastructure? Well, let's dive in.

WASM in Action



We have the hello.js and hello.wasm files from the previous example. Now, let's see how to run them in a cloud environment. To make it more exciting, we'll put this setup in a Docker container, making it easy to move around and use in the cloud.

WASM in Action


```
const express = require('express');
const app = express();
const wasmModule = require('./hello.js');

app.get('hello',(req,res) => {
  wasmModule.onRuntimeInitialized = () => (
    const hello = wasmModule.cwrap('hello', "string, []);
    res.send(hello());
  };
});
app.listen(3000);
```


Now, you might wonder, if WebAssembly can run efficiently on a server-side platform like Node.js, what's stopping us from taking it further? What about the vast cloud infrastructure? Well, let's dive in.

We have the hello.js and hello.wasm files from the previous example. Now, let's see how to run them in a cloud environment. To make it more exciting, we'll put this setup in a Docker container, making it easy to move around and use in the cloud.

First, we set up a Node.js server using Express, listening on port 3000. The highlight is the /hello endpoint. When accessed, it triggers our WebAssembly module, producing the "Hello, WASM in server side!" message. In the cloud, endpoints like this become the gateways for users to interact with our services.

//code//

WASM in Action


```
const express = require('express');
const app = express();
const wasmModule = require('./hello.js');

app.get('hello',(req,res) => {
  wasmModule.onRuntimeInitialized = () => (
    const hello = wasmModule.cwrap('hello', "string, []);
    res.send(hello());
  };
});
app.listen(3000);
```

Comment

WASM in Action

Cloud Server

```
const express = require('express');
const app = express();
const wasmModule = require('./hello.js');

app.get('hello',(req,res) => {
  wasmModule.onRuntimeInitialized = () => (
    const hello = wasmModule.cwrap('hello', "string, []);
    res.send(hello());
  };
});
app.listen(3000);
```

WASM in Action


```
FROM node:14
WORKDIR /app
COPY . .
RUN npm install express
CMD [ "node", "server.js" ]
```


To make our setup cloud-compatible, we used Docker. Our Dockerfile packages the app into a container, ensuring it can run virtually anywhere, especially in cloud environments.

//code//

WASM in Action

```
docker build -t hello-wasm-node-server .
docker run -p 3000:3000 hello-wasm-node-server
```

```
curl https://localhost:3000/hello
```

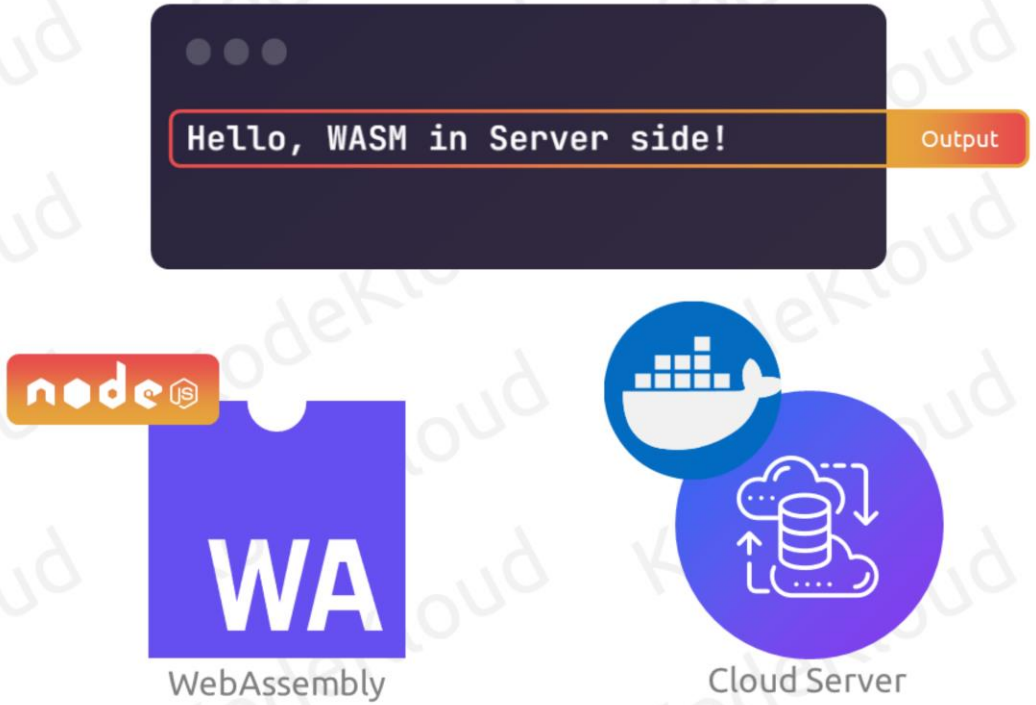

Building and running is straightforward:

//code//

Once everything is set up and running, you can see the output of the WebAssembly module by visiting the /hello endpoint of your server. If you're running everything locally, you'd open a web browser or use a tool like curl in the terminal:

//code//

WASM in Action



This "Hello, WASM in server side" message is the output of the WebAssembly module, executed by the Node.js server inside the Docker container. If you were to deploy this Docker container to a cloud service, you'd access the output using the cloud service's provided URL instead of localhost.
This demonstrates how WebAssembly can work beyond web browsers and be used in the cloud.

WASM in Action

Edge

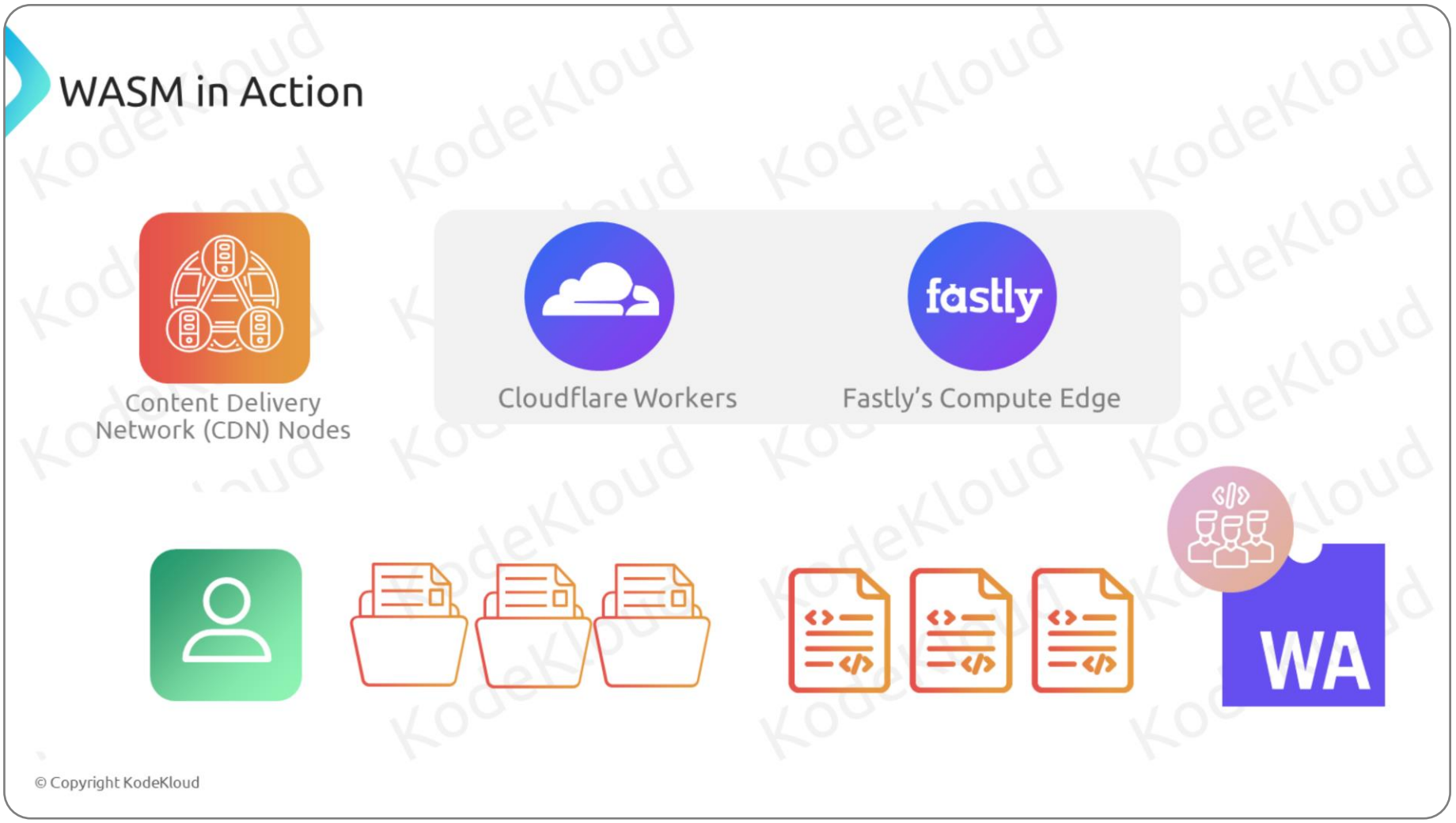
After successfully running our WebAssembly module in a cloud environment using Docker and Node.js, the next logical step is to consider the edge.

WASM in Action

Edge

Content Delivery Network (CDN) Nodes

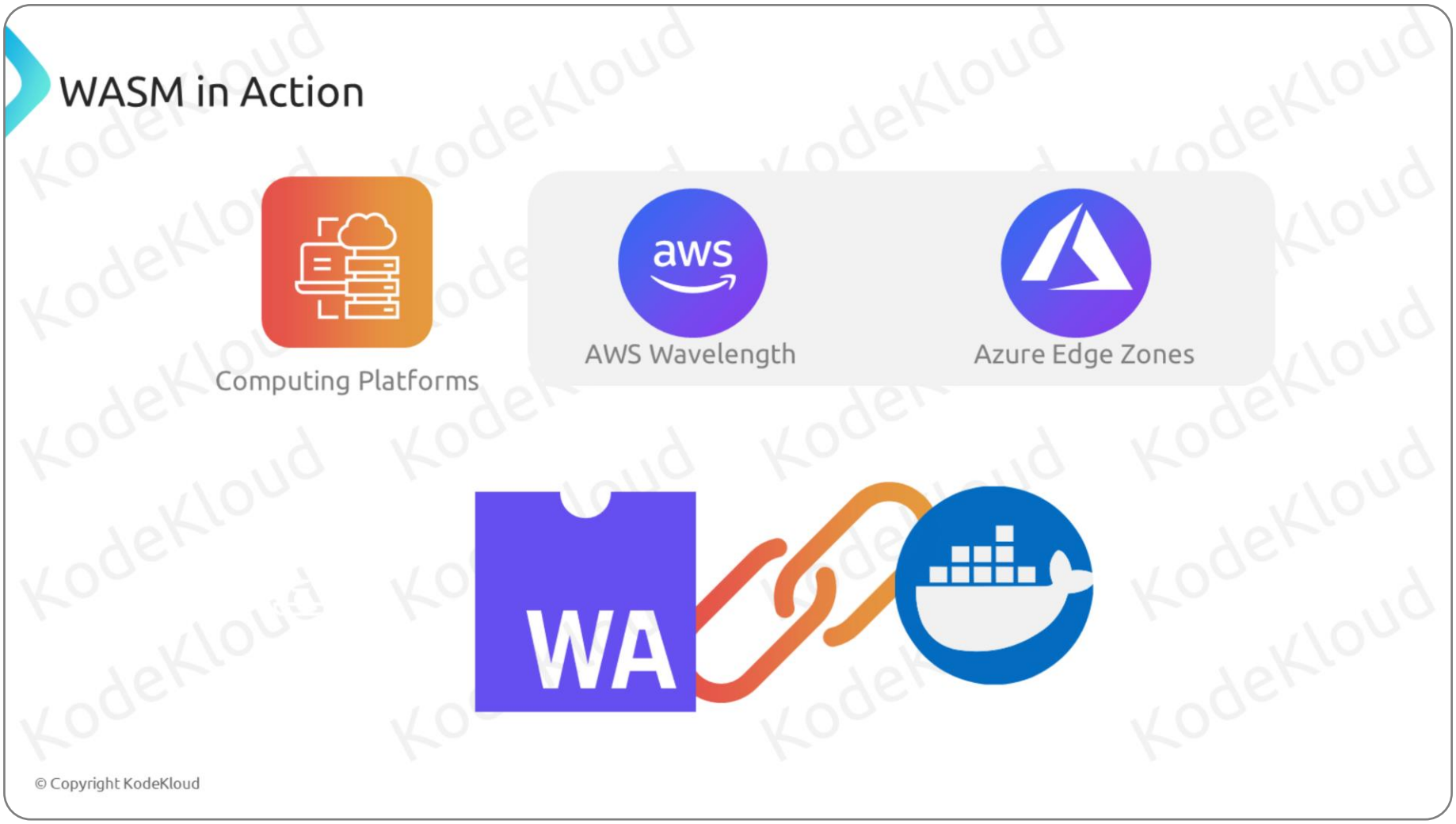
The edge refers to computing resources that are closer to the location of the end-user or data source, often leveraging Content Delivery Network (CDN) nodes or edge devices.
Building on top the previous example and showing how we can use wasm in edge environments.



There are many edge computing platforms, like AWS Wavelength and Azure Edge Zones. Some of these work with containerized apps. As Wasm gets more linked with containers, putting it on these platforms becomes as easy as regular apps.

Throughout this lesson, we've journeyed from the Wasm in the browser to exploring its vast potential in server, cloud and edge environments. We've seen how Wasm's inherent qualities - its efficiency, portability, and security - make it a natural fit

for diverse computing landscapes.



There are many edge computing platforms, like AWS Wavelength and Azure Edge Zones. Some of these work with containerized apps. As Wasm gets more linked with containers, putting it on these platforms becomes as easy as regular apps.

Throughout this lesson, we've journeyed from the Wasm in the browser to exploring its vast potential in server, cloud and edge environments. We've seen how Wasm's inherent qualities - its efficiency, portability, and security - make it a natural fit

for diverse computing landscapes.

WASM Runtimes

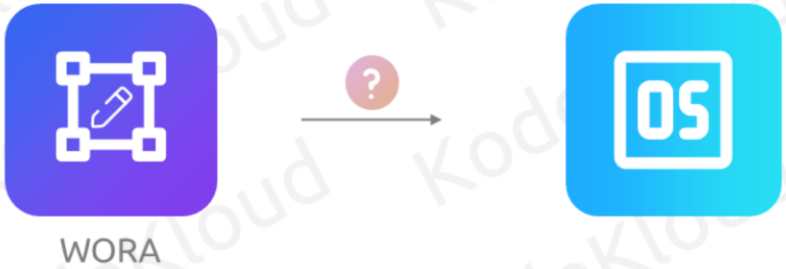
Introduction

Remember in our last lesson We found out WebAssembly is not only for web browsers. There are these things called "runtimes" that let WASM work in other places too, like on big servers in the cloud or on small devices we use daily.

Understanding WASM Runtimes

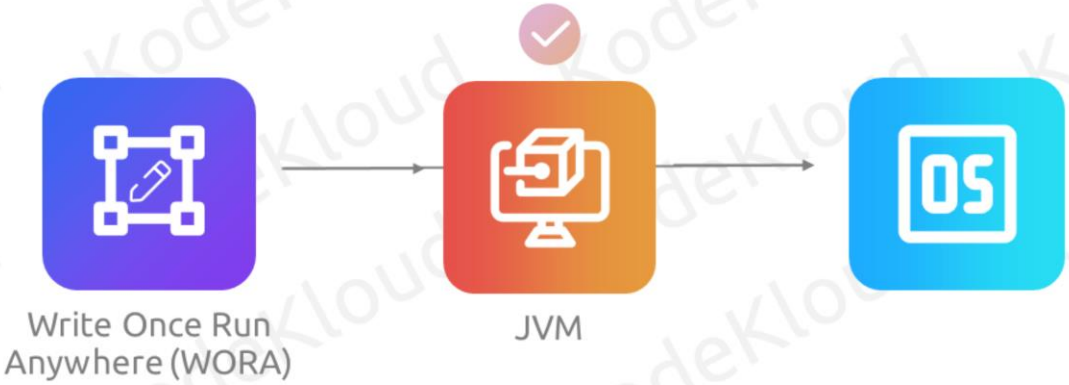
Now, in this lesson, we're diving deep into these WASM runtimes. We'll explore what they are, how they work, and why they're so crucial for letting WASM shine everywhere, not just on the web.

Understanding WASM Runtimes



You might have come across the term "Write Once, Run Anywhere" or WORA. This was Java's big promise. The idea was simple: write your code once, and it should run on any device or platform that has the Java Virtual Machine (JVM). The JVM is Java's runtime environment. It's like a special translator that takes the Java code you write and turns it into something your computer understands. So, no matter where you're running your Java program - be it a Windows PC, a Mac, or even a tiny device - as long as it has a JVM, it understands the code.

What is WASM runtimes



Exploring Runtimes

Now, imagine taking that idea of "write once, run anywhere" and applying it to the web. That's where WebAssembly (Wasm) comes in.

Exploring Runtimes

WASM

JAVA bytecode

Wasm is a bit like Java's bytecode, but for the web. As you already know, it's a new type of code that can be run in modern web browsers, and it's super fast.

Exploring Runtimes

But here's the cool part: Wasm isn't just for browsers as we said in our last lesson. With the right tools, called runtimes, Wasm code can run outside the browser too. These runtimes are like the JVM for Java but designed for Wasm. They take Wasm code and execute it on different platforms.

Exploring Runtimes

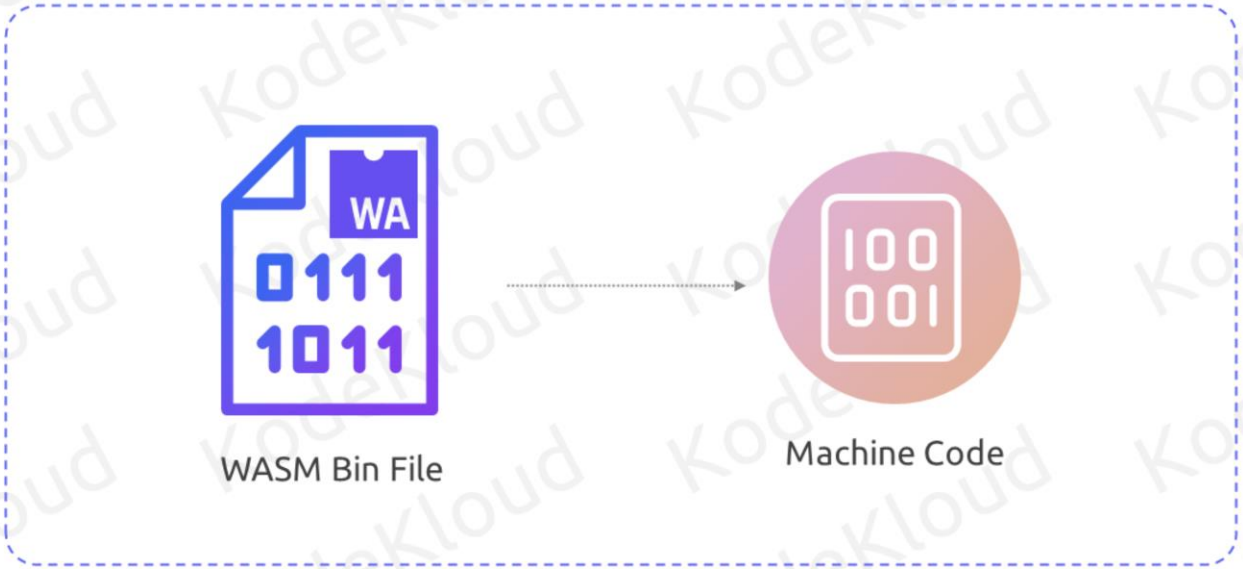
Exploring Runtimes

Exploring Runtimes

Let’s take a moment to understand what exactly is a WASM Runtime and how it works.

Imagine you have a WebAssembly module. As you have already known, this module is like a set of instructions, packaged in a binary format. It is created from higher-level languages like C, C++ or Rust.

Exploring Runtimes



Now you bring this mysterious thing called WASM Runtime. This runtime could be your web browser as you have seen in earlier lessons, or a standalone environment on a server.

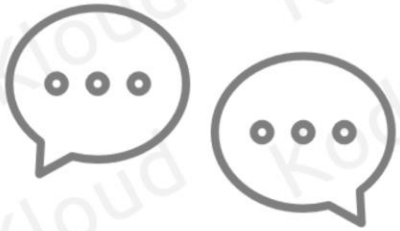
This runtime acts like a welcoming committee. It takes this module and starts to unpack it, readying it for execution.

Next, the WASM Runtime translates this module into something that the host machine understands. This process can be like

translation a foreign language into your native tongue.

Exploring Runtimes

"Ahead of time"
(AOT)



"Just in time"
(JIT)

Depending on the runtime, this translation can be done ahead of time (AOT), just like preparing a speech on advance, or just in time (JIT), like translating on the fly during a live conversation. These different techniques have been discussed in our previous lessons.

Exploring Runtimes

Now you know that these translation techniques are taken care of by this thing called “WASM Runtime”.

Once translated, the instructions are ready to be executed. The runtime ensures that these instructions are run efficiently and correctly. During this phase, the runtime manages resources like memory, ensuring the WASM code has what it needs while keeping it isolated and secure.

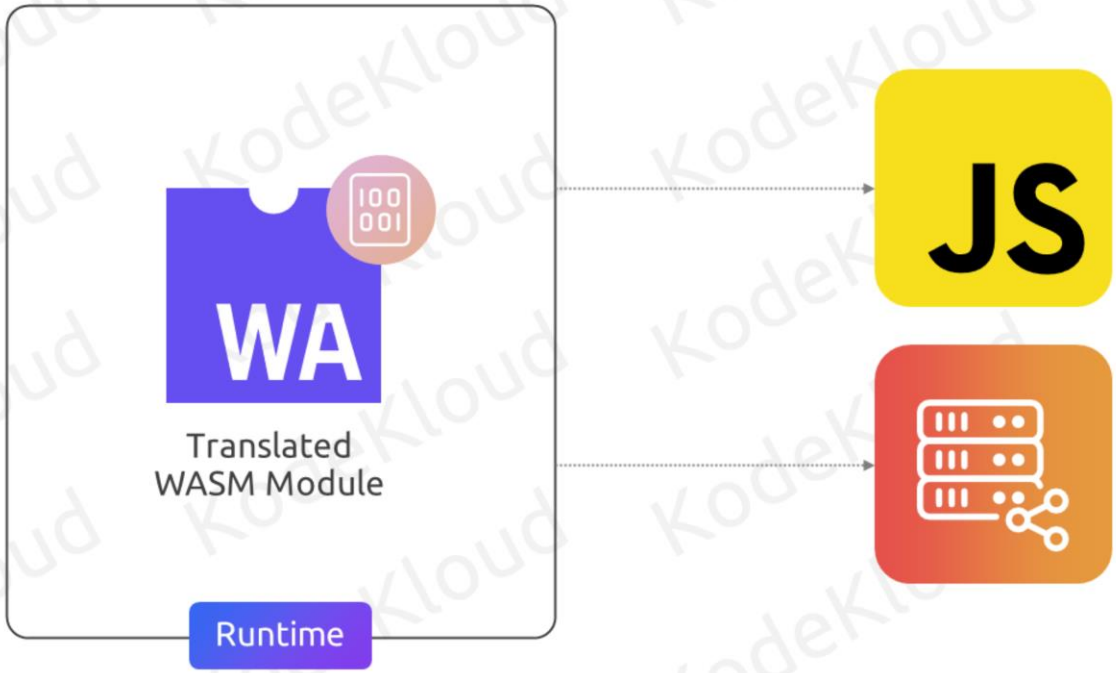
IF the WebAssembly module needs to interact with the outside world, like calling JavaScript functions in a web browser or accessing system resources in a server environment, the run time manages these interactions.

It’s like a mediator, ensuring that these interactions are safe and follow the rules set by the WebAssembly and the host environment.

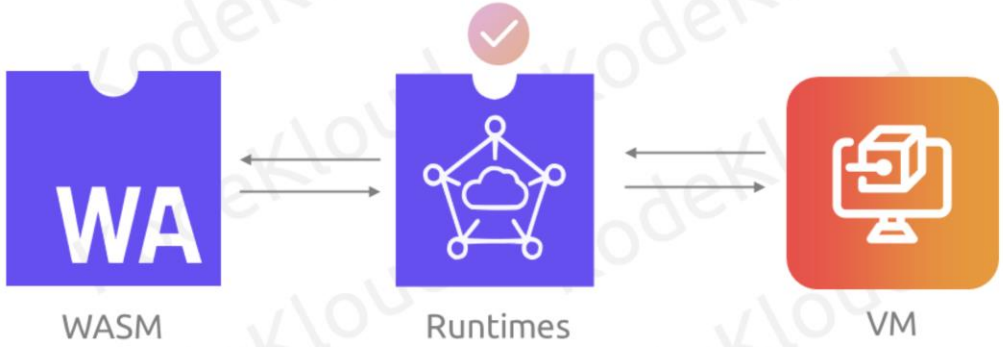
After executing all the instructions, the runtime concludes the process. The results of the WebAssembly module’s execution are then available to the host environment or the end user.

Exploring Runtimes

Exploring Runtimes

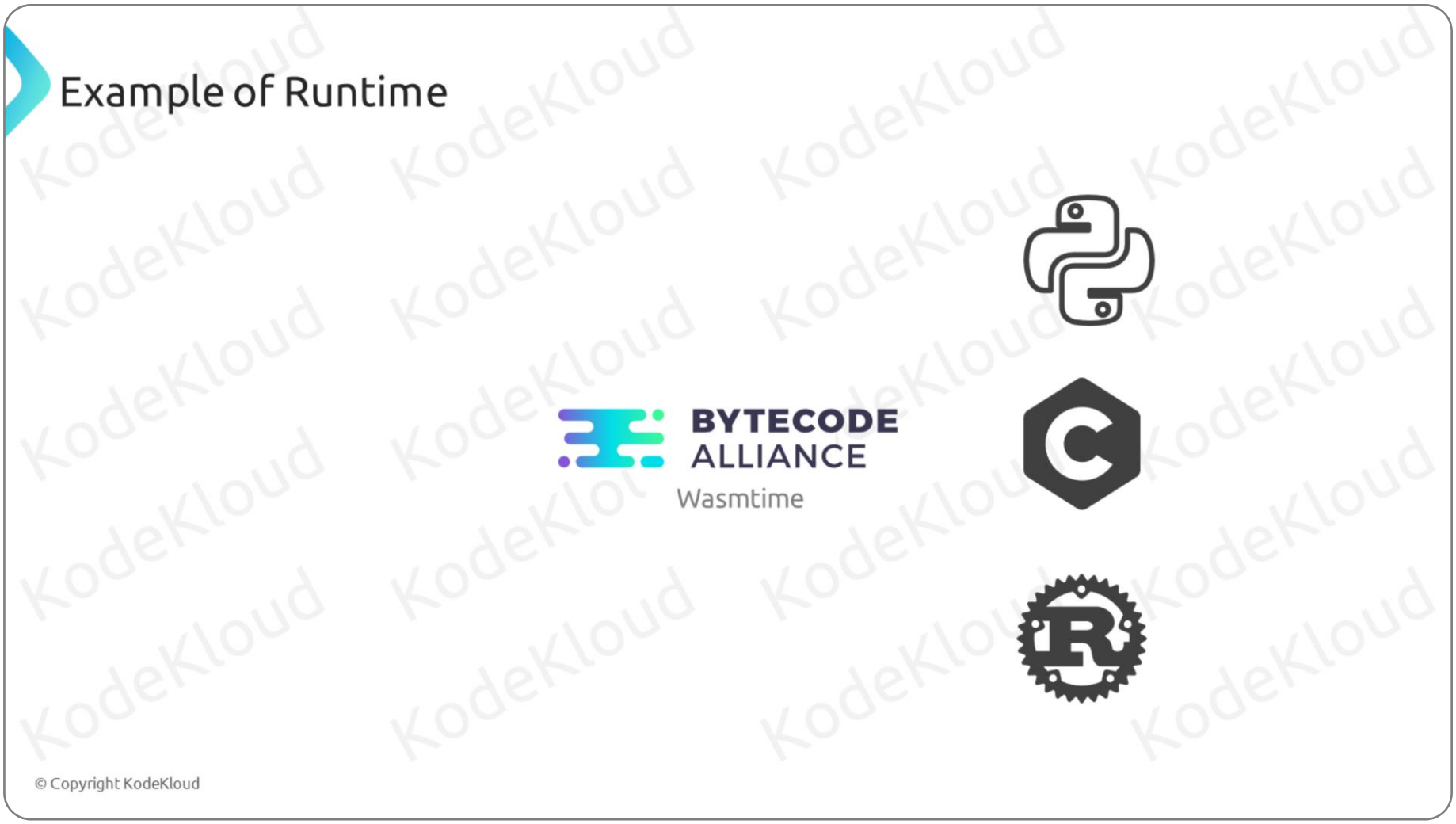


Wasm Runtime Role



The role of the Wasm runtime is similar to a stage manager in a play. It sets the stage, ensuring everything is in place for the performance. When you have some Wasm code, you hand it over to the runtime. The runtime then gives this code to the VM to execute. If you have specific tasks or functions in the Wasm code, the runtime ensures the VM performs them. Once the VM completes its tasks, the runtime makes sure you get the results. And when the show's over, the runtime ensures everything is wrapped up neatly, shutting down the VM.

In essence, the Wasm runtime is ensuring that the WebAssembly code not only runs correctly but also delivers the expected results wherever you want it to.



Now that we have a grasp of what a Wasm runtime is, let's delve into some popular examples and see how they fit into the bigger picture of the Wasm ecosystem.

First up, we have Wasmtime, a WebAssembly runtime known for its adaptability. It can be embedded in languages like Rust, Python, and C. What sets it apart is its ability to scale seamlessly, working well on both small IoT devices and robust servers. Wasmtime offers flexible configurations, including support for pre-compilation and runtime interpretation. Its compatibility

with Cranelift ensures rapid execution of code.

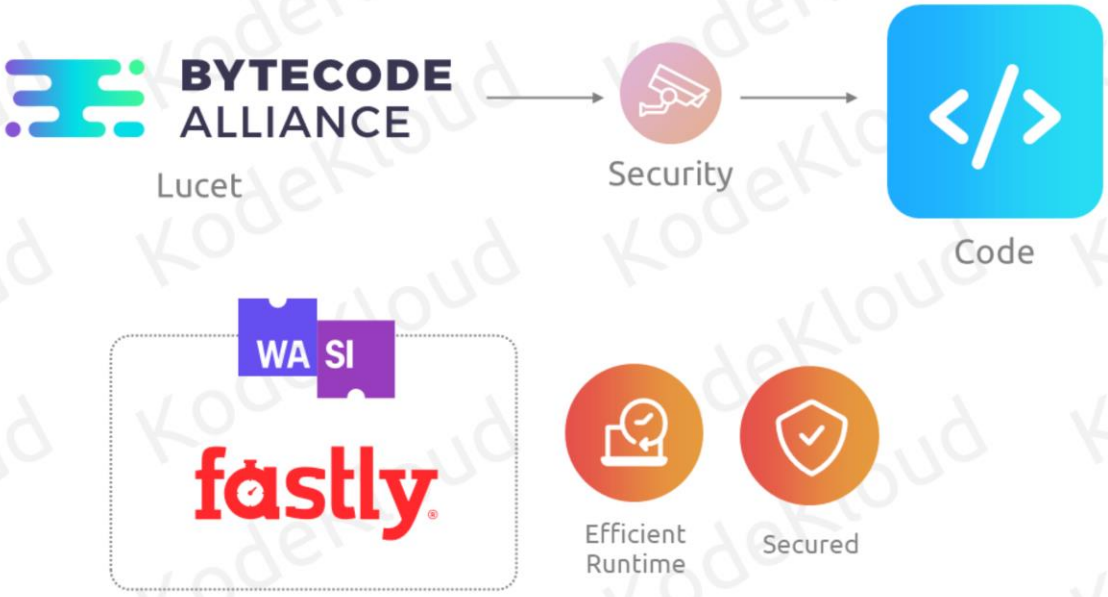
Example of Runtime

Next in the lineup is Wasmer, which specializes in lightweight containers for WebAssembly applications. What makes Wasmer stand out is its universal compatibility. It can be deployed on various platforms, from cloud services to desktops and IoT devices. Wasmer's unique feature is its capability to seamlessly integrate into multiple programming languages, enabling easy adoption of WebAssembly across the development landscape. Additionally, it boasts impressive execution speeds, rivaling native code.

Example of Runtime

Moving on, we have Lucet, a WebAssembly runtime designed with security in mind. Lucet is perfect for developers who need to run potentially risky WebAssembly programs securely. As part of the Bytecode Alliance project, Lucet executes untrusted code across different platforms with near-native speeds. It excels in handling the WebAssembly System Interface, especially in environments like the Fastly edge cloud. Moreover, it maintains an efficient runtime footprint without compromising on security.

Example of Runtime



Moving on, we have Lucet, a WebAssembly runtime designed with security in mind. Lucet is perfect for developers who need to run potentially risky WebAssembly programs securely. As part of the Bytecode Alliance project, Lucet executes untrusted code across different platforms with near-native speeds. It excels in handling the WebAssembly System Interface, especially in environments like the Fastly edge cloud. Moreover, it maintains an efficient runtime footprint without compromising on security.

Example of Runtime

Microcontroller

Server

Wasm3

Another runtime Known for its incredible speed is wasm3. wasm3 is a high-performance runtime. It's designed to be portable and works on a range of platforms, from microcontrollers to servers.

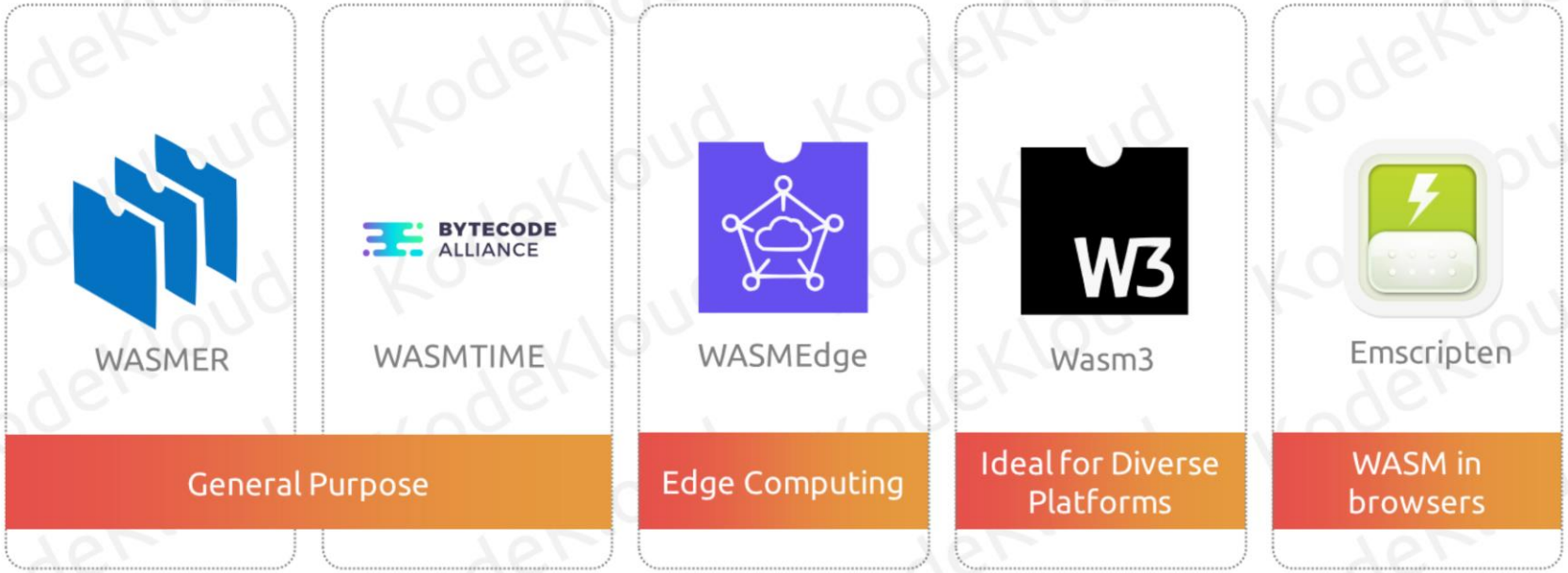
Last but not least, let's talk about WAMR. WAMR's minimalistic design makes it an ideal choice for resource-constrained devices. At its core, it features the 'iwasm' VM, supporting just-in-time (JIT) and ahead-of-time (AOT) compilation, as well as WebAssembly interpretation. Despite its compact size, WAMR delivers performance close to native speeds. It adheres to W3C WASM MVP standards, ensuring consistent and reliable execution.

Each of these runtimes plays a unique role in the Wasm ecosystem. While Wasmtime and Wasmer are general-purpose

runtimes suitable for a range of applications, WASMEdge is tailored for edge computing. On the other hand, wasm3 is all about performance and portability, making it ideal for diverse platforms. Emscripten bridges the gap between traditional programming languages and the web, bringing C and C++ to browsers via WebAssembly.

So, understanding these runtimes and their unique offerings helps in making informed decisions based on the requirements of specific projects or applications.
The future of WebAssembly and its runtimes is promising. As the technology landscape evolves, these runtimes will continue to adapt, bringing forth new features and optimizations

Role of Runtimes



Each of these runtimes plays a unique role in the Wasm ecosystem. While Wasmtime and Wasmer are general-purpose runtimes suitable for a range of applications, WASMEdge is tailored for edge computing. On the other hand, wasm3 is all about performance and portability, making it ideal for diverse platforms. Emscripten bridges the gap between traditional programming languages and the web, bringing C and C++ to browsers via WebAssembly.

Role of Runtimes

So, understanding these runtimes and their unique offerings helps in making informed decisions based on the requirements of specific projects or applications.
The future of WebAssembly and its runtimes is promising. As the technology landscape evolves, these runtimes will continue to adapt, bringing forth new features and optimizations

Role of Runtimes

Introduction to Cloud Native WASM

Cloud-Native – Concept

In our previous lesson, we explored the vast potential of WebAssembly (WASM) beyond the confines of the browser.

Cloud-Native – Concept

Cloud-Native Apps

We ventured into server-side runtimes, cloud environments, and even touched upon the edge with CDNs.

Cloud-Native – Concept

We saw it work on servers like Node.js and even in cloud setups using Docker. It's like WASM can fit anywhere, not just on our web browsers.

Cloud-Native – Concept

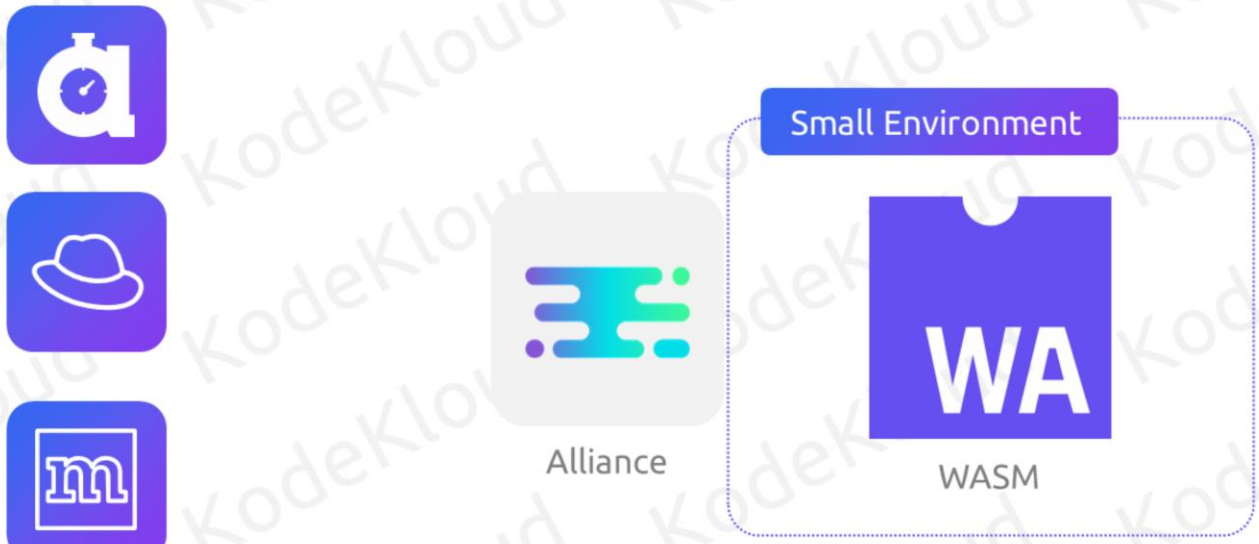
Alright, let's talk about "cloud-native." It's like building a house specifically for a beach instead of the city. The house is designed to handle the sand, the waves, and the sun. Similarly, cloud-native apps are made especially for the cloud. They can grow when needed, fix themselves if something goes wrong, and adjust to changes quickly.

As we learnt, WASM is fast, safe, and can work anywhere. These things make it perfect for the cloud. Cloud-native is all about being quick, flexible, and reliable. So, when you put WASM in a cloud-native setup, it's like adding a turbocharger to a sports car. They just work really well together.

Cloud-Native – Concept

When we mix WASM with cloud-native, we get "Cloud Native WASM." It's like taking the best parts of both and making something even better. With this, our apps don't just live in the cloud; they become part of it. They can handle more users, fix issues on their own, and respond faster.

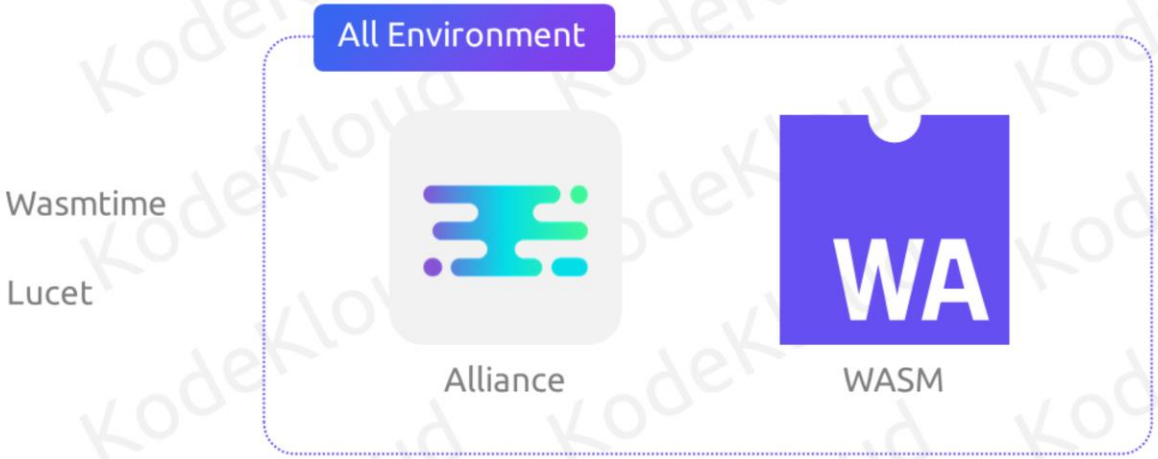
Adapted Cloud-Native WASM and Benefits



To truly understand the impact of Cloud Native WASM, let's look at some big names that have adopted this approach and the benefits they've gained.

The Bytecode Alliance is a group of industry leaders, including Mozilla, Fastly, Intel, and Red Hat, working together to push the boundaries of WebAssembly. Their focus is on creating a secure foundation for building applications that work anywhere, from browsers to cloud-native environments.

Adapted Cloud-Native WASM and Benefits



The alliance is actively developing tools and runtimes like Wasmtime and Lucet, which are designed to run WASM outside the browser, especially in cloud-native setups.

Adapted Cloud-Native WASM and Benefits

Fastly

Krustlet

Fastly's Computer@Edge platform. This is a system from Fastly that uses WebAssembly (WASM) to run programs near where users are. This makes delivering content like web pages faster.

The Krustlet us another tool that works with Kubernetes, a popular system for managing cloud applications. Krustlet allows Kubernetes to run tasks using WebAssembly, which is a newer way of running programs that's gaining popularity and support.

How WASM fits with Cloud-Native Architecture's core pillars

Cloud-native is more than just a buzzword; it's a philosophy that focuses on building and running scalable applications in dynamic environments like public, private, and hybrid clouds. WebAssembly (WASM) plays a pivotal role in enhancing this philosophy. Let's delve into the core pillars of cloud-native and see how WASM fits in:

How WASM fits with Cloud-Native Architecture's core pillars

Application

Instead of constructing a single, monolithic application, in cloud-native architecture, the application is divided into smaller, independent services.

How WASM fits with Cloud-Native Architecture's core pillars

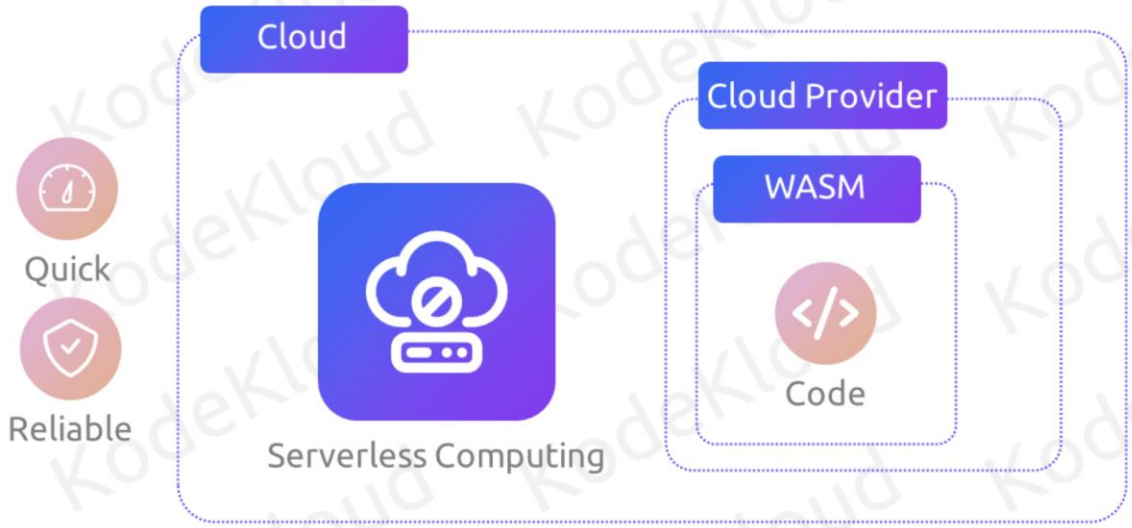
Each service handles a specific function and can be developed, deployed, and scaled on its own. WASM enhances this by offering a consistent runtime for these services, ensuring they run efficiently and securely across different environments.

Cloud-Native – Core Pillars

Containers are lightweight, standalone packages that contain everything needed to run a piece of software, including the code, runtime, system tools, and libraries. WASM complements containerization by providing a compact binary format that can be embedded within containers. This ensures that applications within containers benefit from the fast execution and security sandboxing features of WASM, making them even more efficient and resilient.

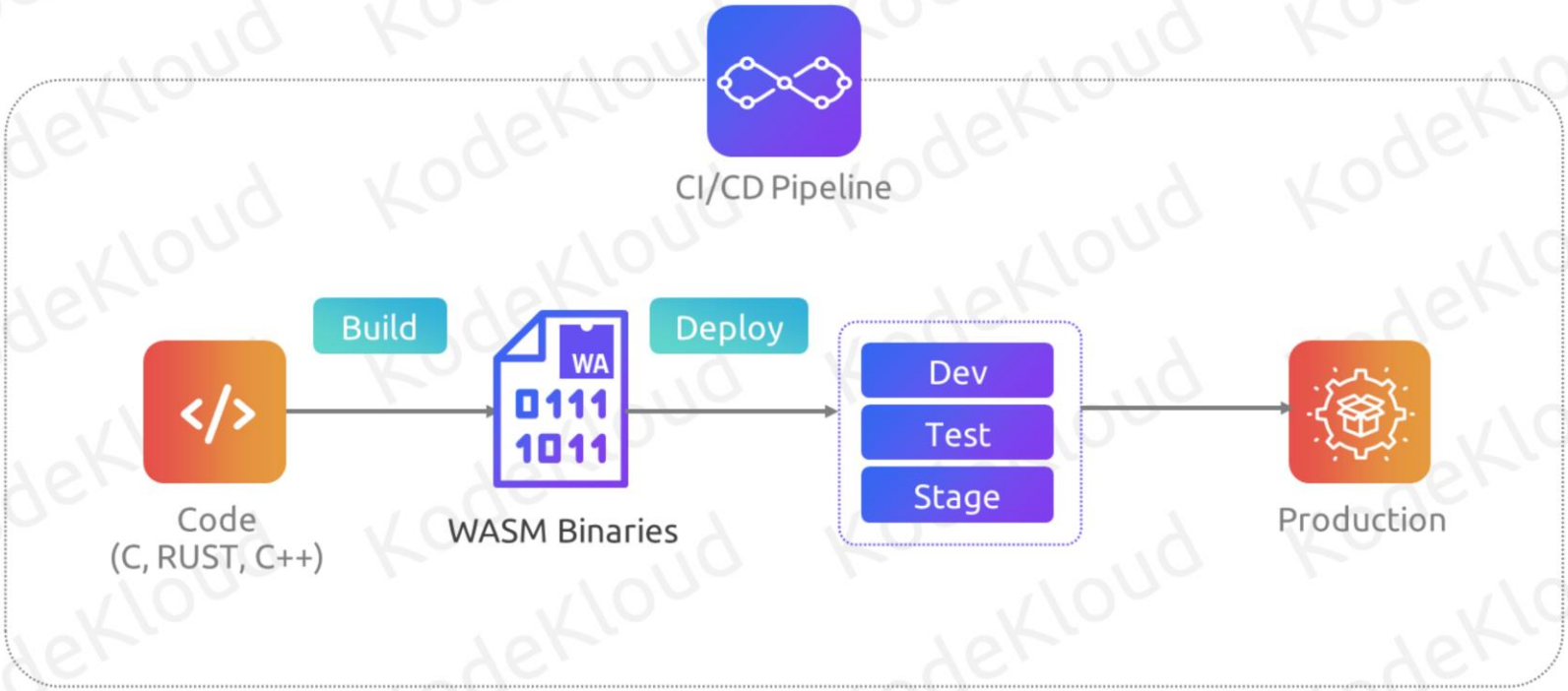
Cloud-Native – Core Pillars

Cloud-Native – Core Pillars



Serverless computing allows developers to build and run applications without thinking about the underlying infrastructure. With WASM, serverless functions can be written in multiple languages, compiled to WASM, and then executed in a consistent environment provided by the cloud provider. This means developers can focus solely on their code, knowing that it will run efficiently and securely, while the cloud provider handles the infrastructure, scaling, and management.

Cloud-Native – Core Pillars



CI/CD emphasizes the importance of integrating changes continuously and ensuring that they can be deployed to production without manual intervention. WASM fits seamlessly into CI/CD pipelines. As developers push code changes, they can be automatically compiled to WASM binaries, tested in consistent environments, and deployed. This ensures that applications benefit from the rapid iteration of CI/CD while also leveraging the portability and performance of WASM.

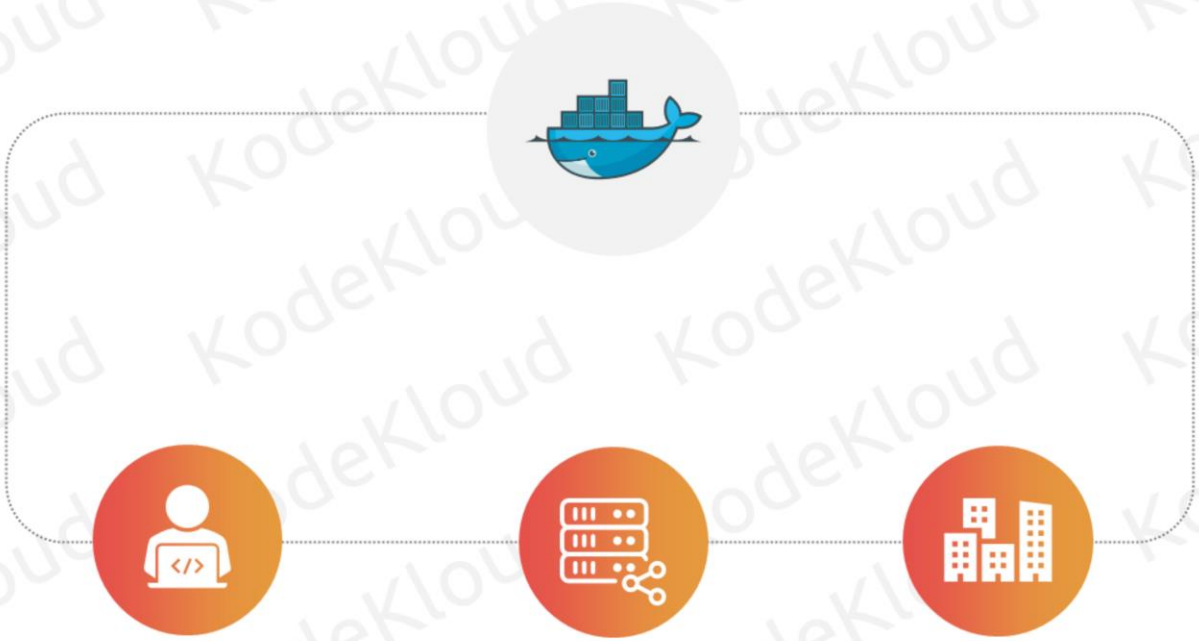
With these in mind, in the upcoming lessons, we'll deep dive into how WASM can enhance microservices, bring new possibilities to serverless computing, and integrate seamlessly with containerization technologies.

Exploring WebAssembly (WASM) and Docker

Docker

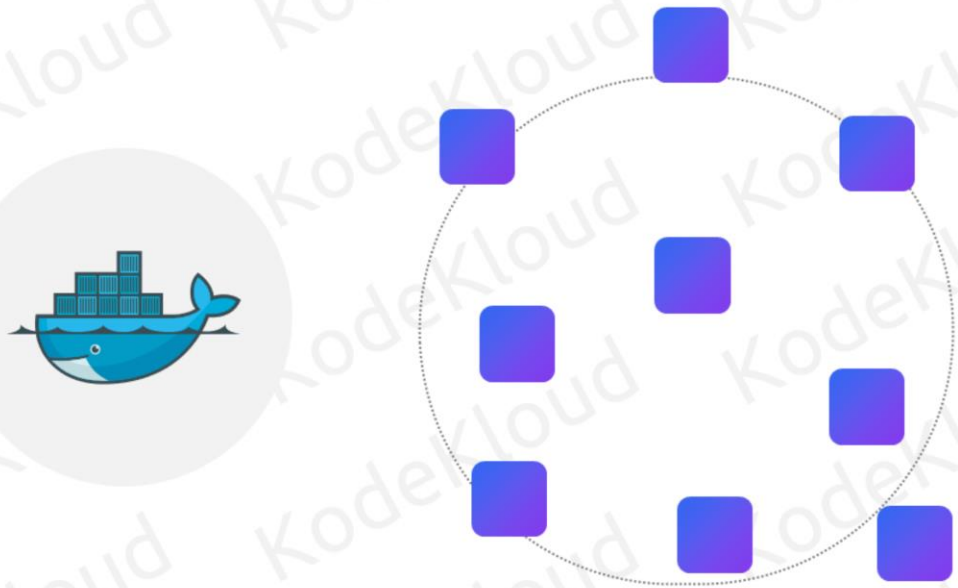
Docker; As you know, Docker really changed the game in how we deploy software.

Docker



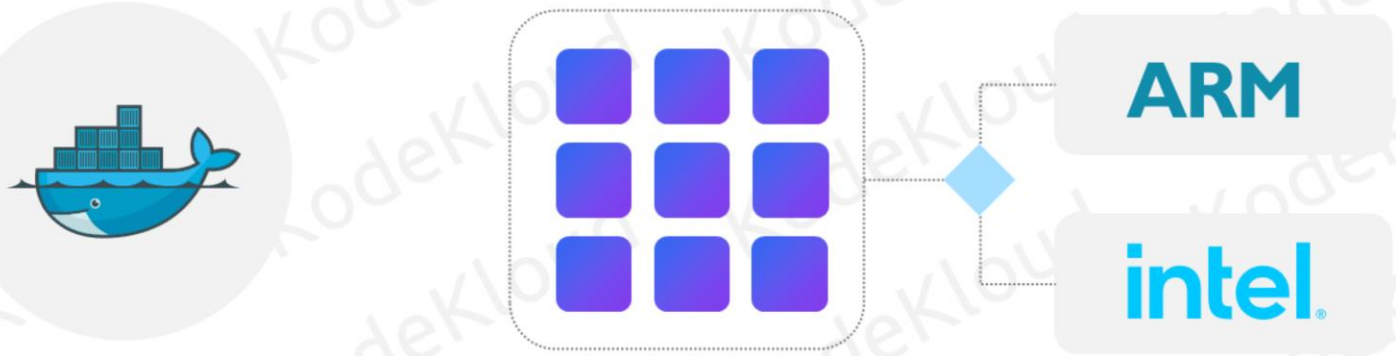
It solved a lot of problems with making sure applications run the same way in different places, like on a developer's laptop or a big server in the cloud.

Docker

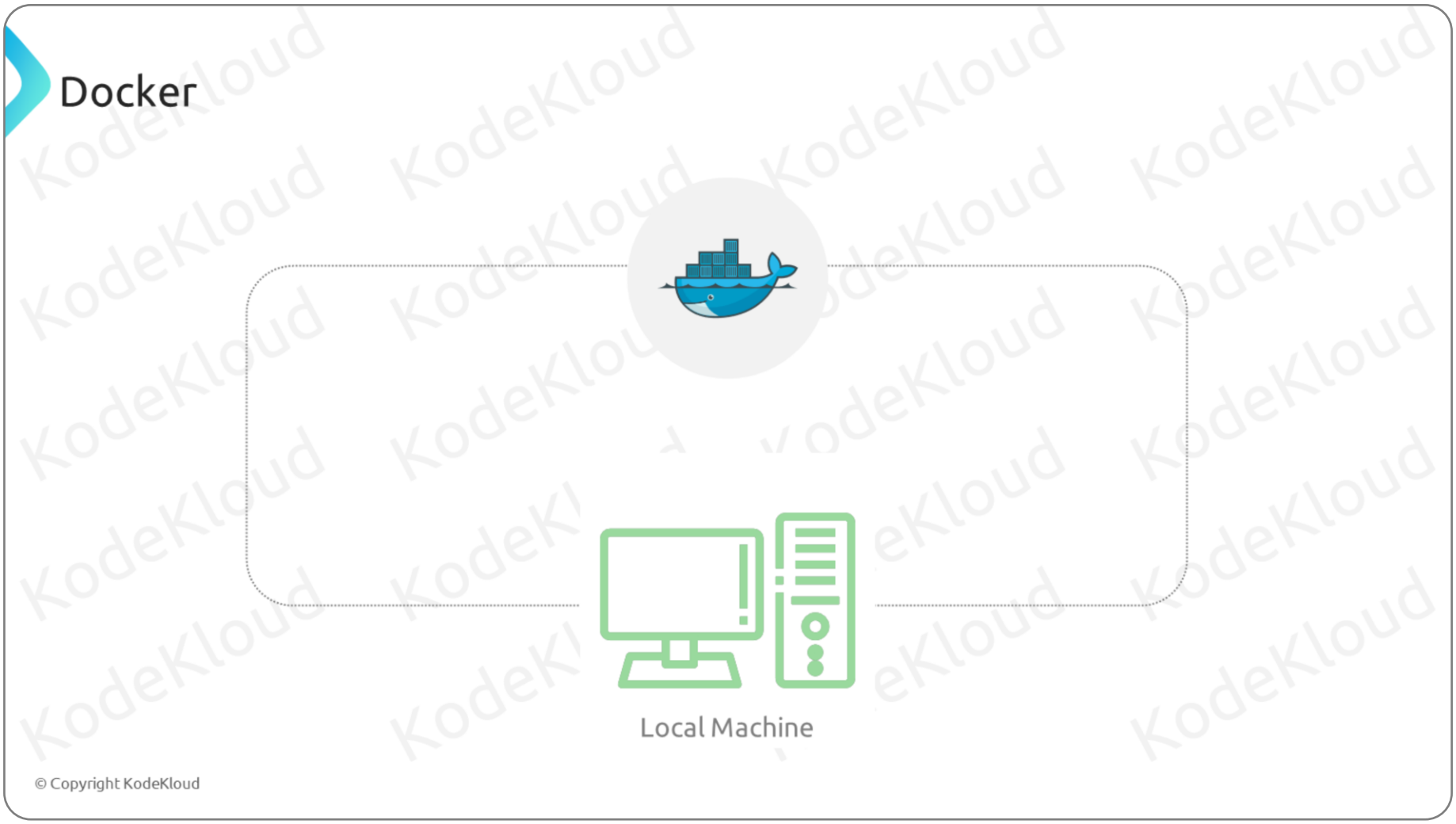


However, Docker isn't perfect and has its own set of issues, especially when it comes to the type of computer system it runs on.

Docker

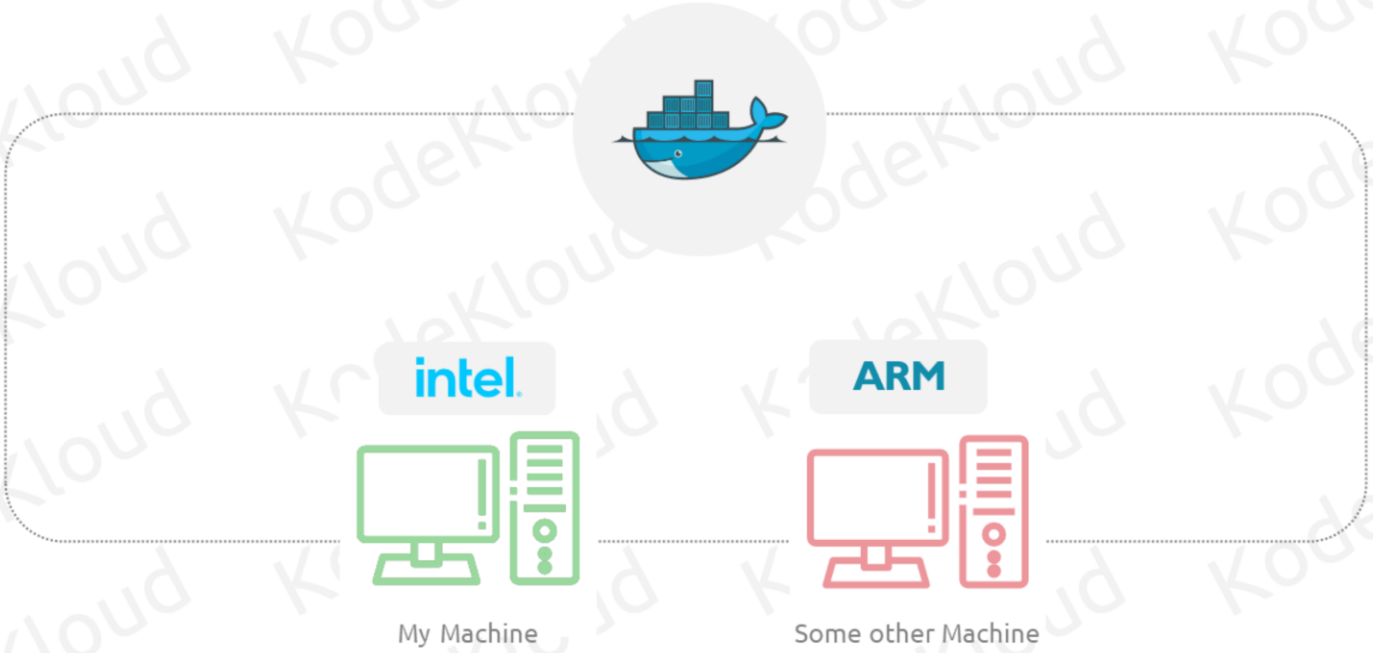


Even though Docker packs everything an application needs into one package, it still depends on the computer's architecture like Intel or ARM.



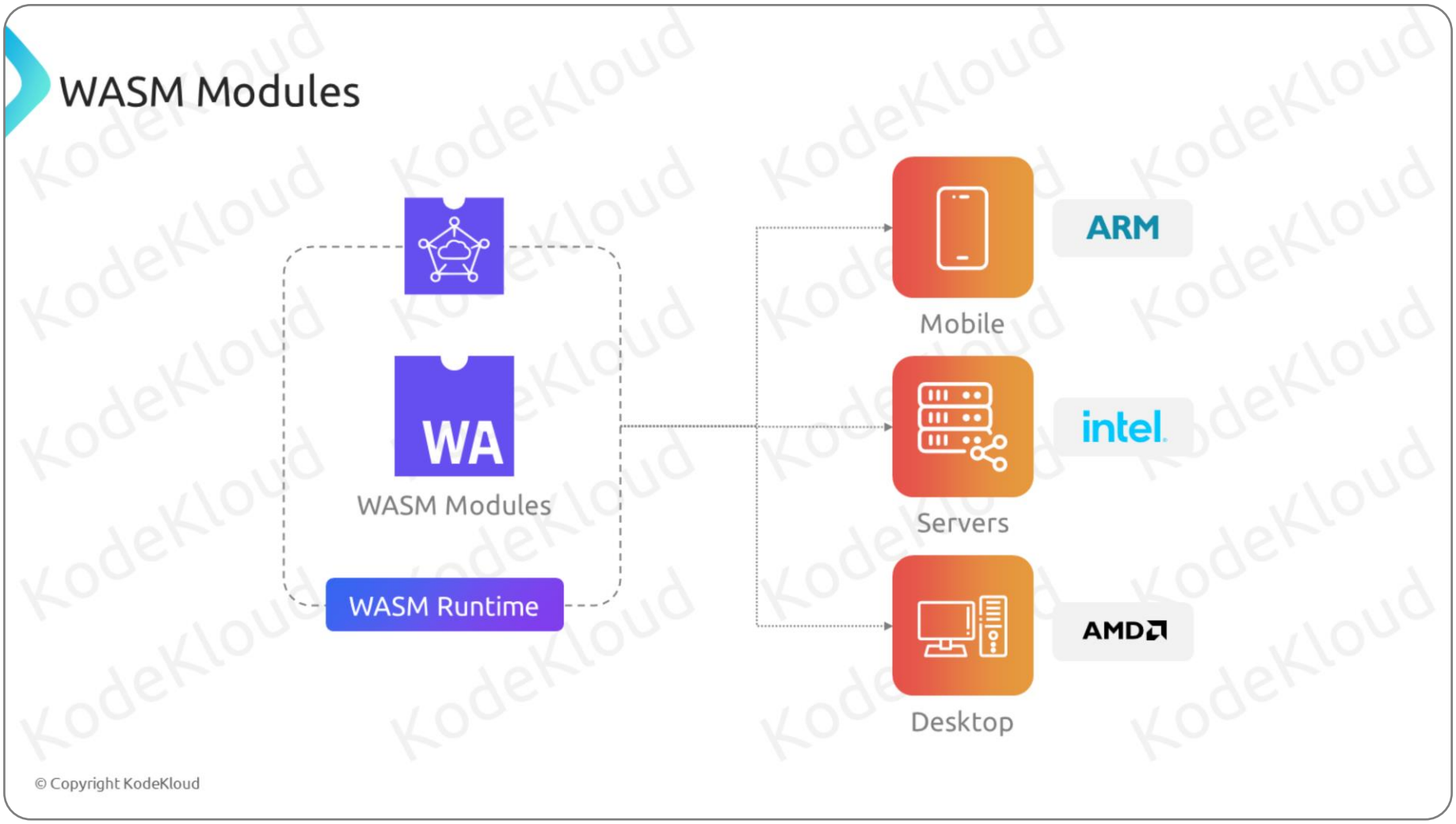
So, a Docker image made for one kind of system might not work right on another, leading to the familiar problem of "it works on my machine" but not elsewhere.

Docker



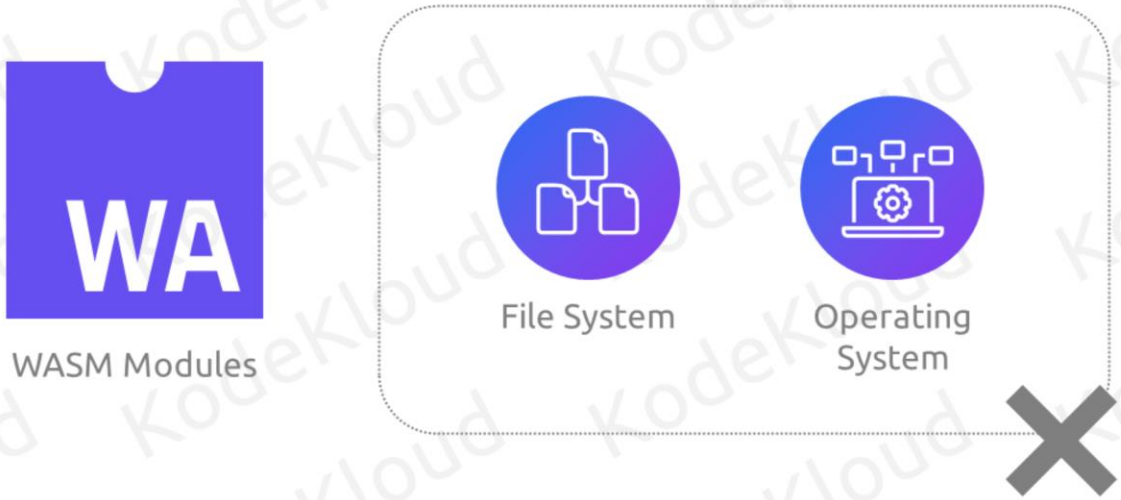
"It works on my machine, not elsewhere!!"

WASM modules are pre-compiled binaries not tied to a specific system architecture.



This means they can run on any platform that has a suitable WebAssembly runtime, such as Wasmtime or Wasmedge, regardless of whether it's Intel, ARM, or AMD.

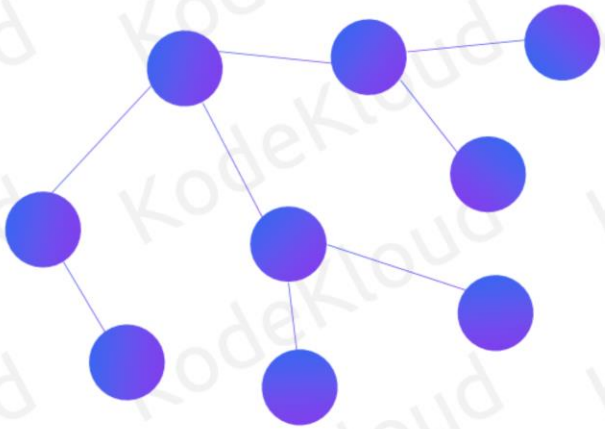
WASM Modules



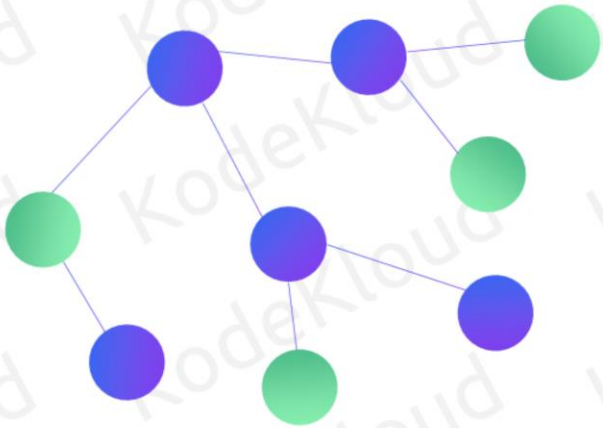
WASM works differently because it doesn't need the whole file system or things specific to an operating system. Instead, it focuses on just using what's necessary when the program is running.

WASM Modules

WebAssembly System Interface (WASI)

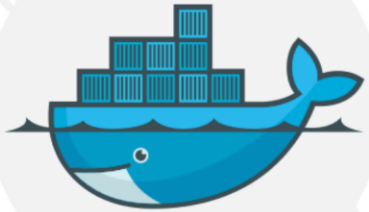


WASM Modules



This is done through something called the WebAssembly System Interface (WASI), which we've already learned about. Basically, WASI lets WASM programs access only the essential parts they need to run, making things simpler and more efficient.

Docker and WASM



After understanding Docker's architecture-centric approach and WASM's platform-agnostic capabilities, let's explore these two technologies based on various critical factors with a real world example.

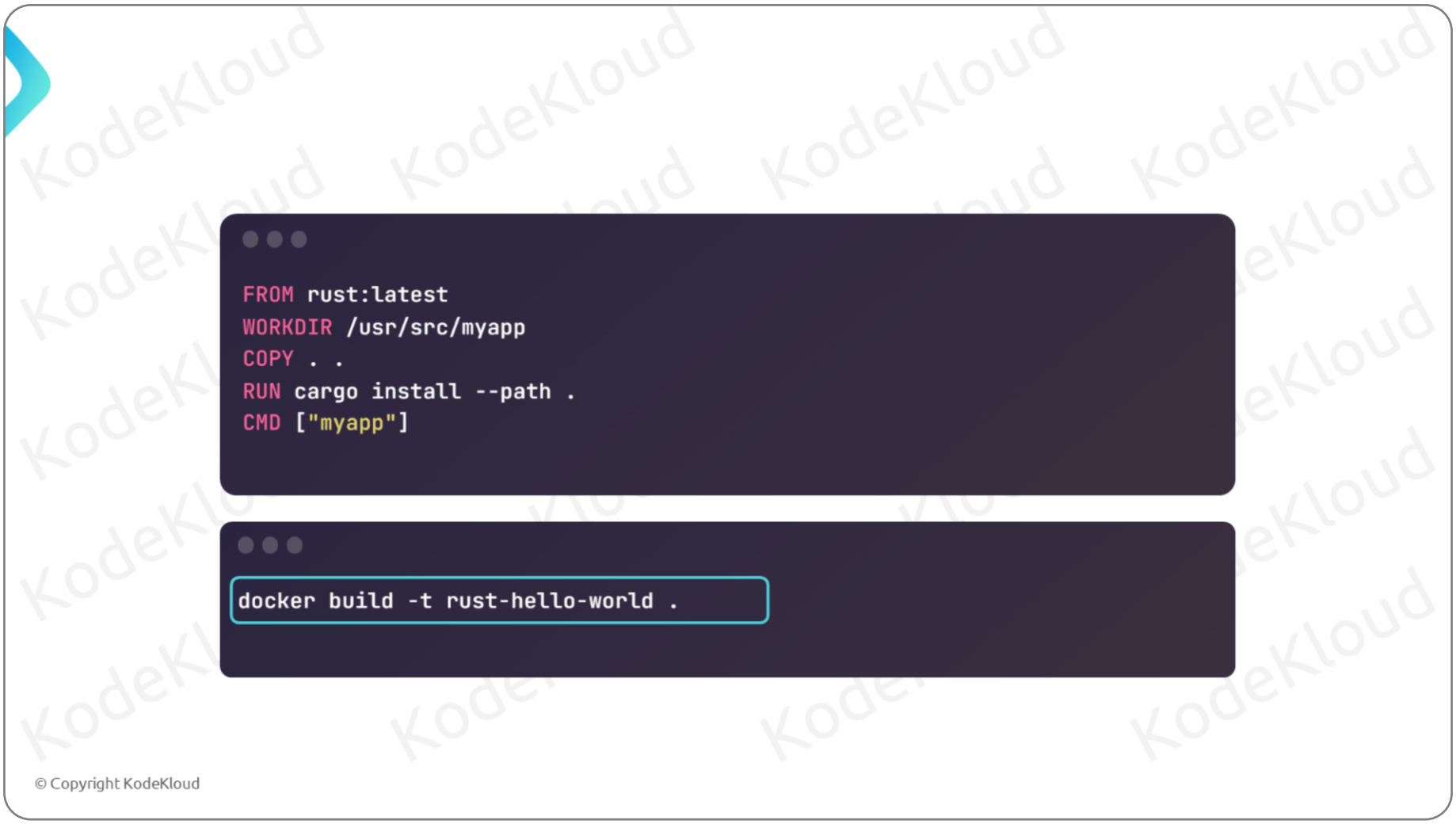
Docker and WASM


```rust
use warp::Filter;

#[tokio::main]

async fn main() {
    let route = warp::path!("hello" / "world").map(|| "Hello, World!");
    warp::serve(route).run(([127,0,0,1], 3030)).await;
}
```


We are going to write a simple Rust program to create a web server. This server will respond with "Hello, World!" when accessed.

Docker and WASM

Next, we containerize this Rust server using Docker. We'll need a Dockerfile to define how the Docker image should be built.



Next, we containerize this Rust server using Docker. We'll need a Dockerfile to define how the Docker image should be built.

// code //

Finally, we will run the docker build command to create the docker image.

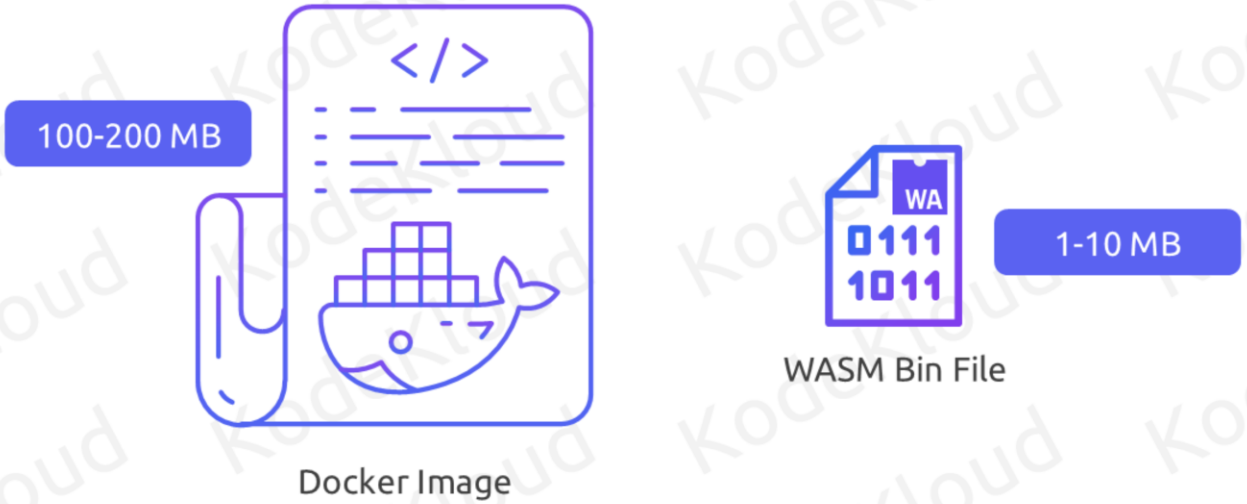
// code //

Next we compile this Rust program to a WASM binary. This involves setting up the Rust toolchain for WASM compilation as we have done in a previous demo.

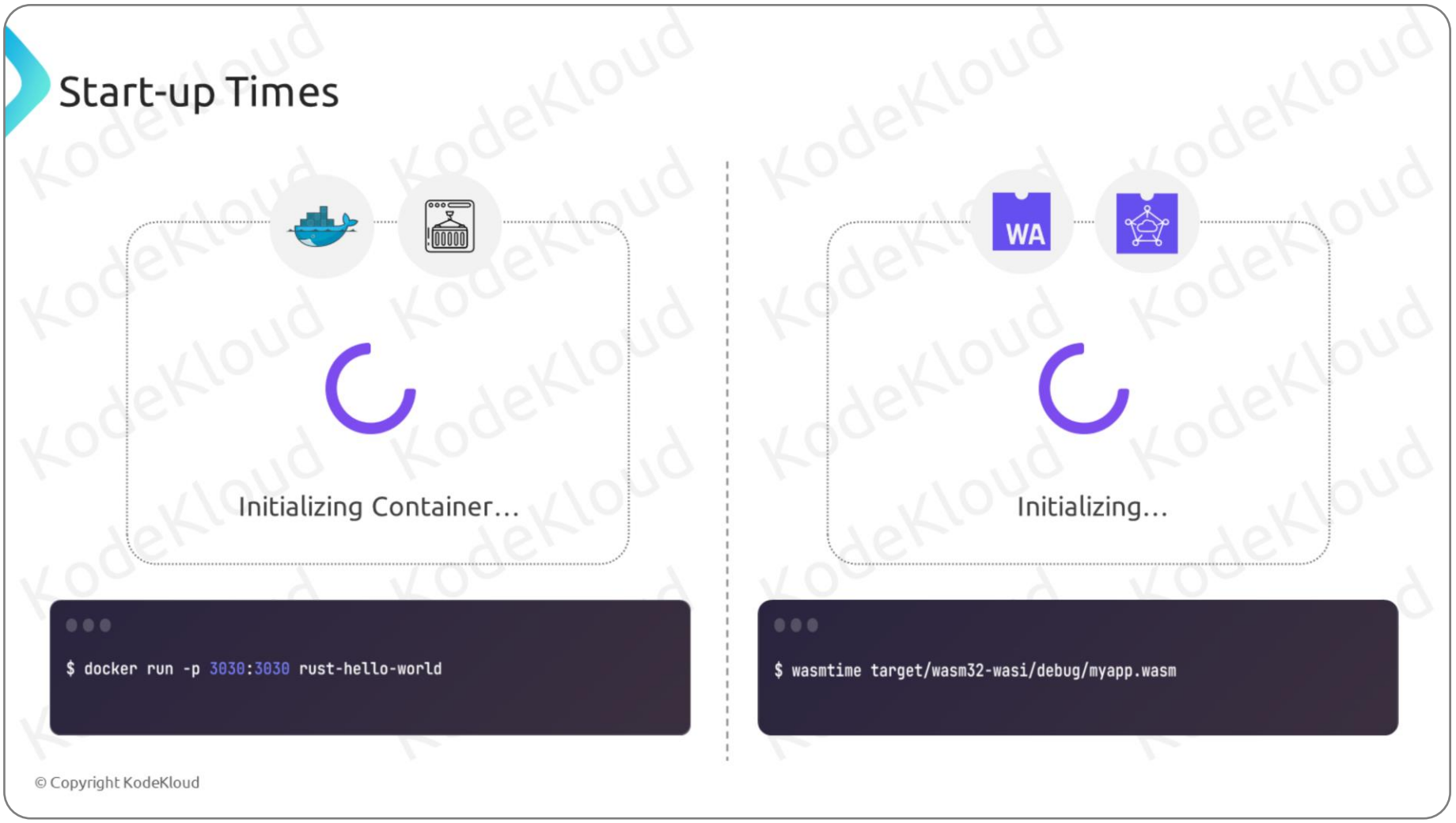
Then, we will compile the rust program to a wasm binary with the cargo build command.

// code //

Size of the final package



Now, if you compare the sizes of the Docker image and the WASM binary.
Typically, for a basic Rust server, the Docker image size can be around 100-200 MB.
The WASM binary, however, is much smaller, likely a few 1-10 MBs in size.



Let's start the Docker container by running something like docker run -p 3030:3030 rust-hello-world.

// code //

It might take a few seconds to start up, due to setting up the virtual environment.

While it is starting, Let’s try to run the wasm binary module.
To run the WASM module, we can use a WASM runtime like Wasmtime as learnt previously.

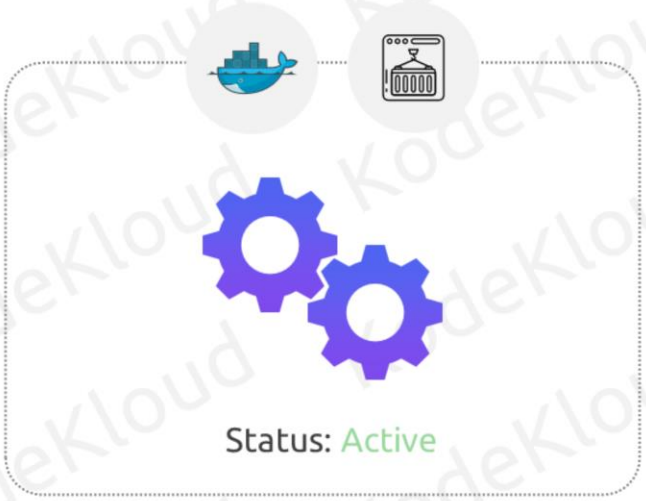
You'd execute a command like
wasmtime target/wasm32-wasi/debug/myapp.wasm.

// code //

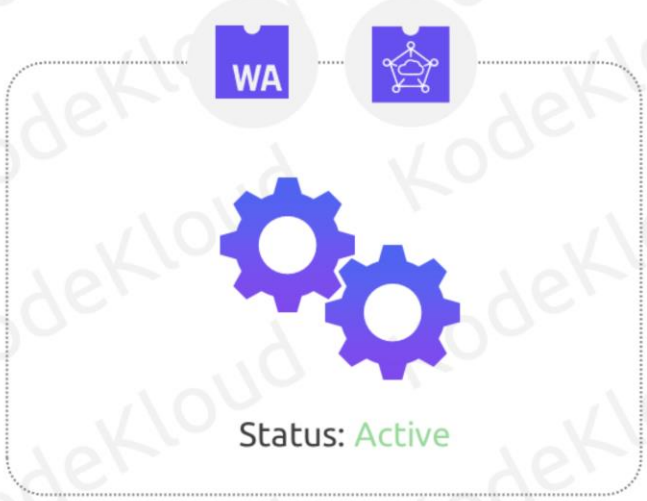
The startup here is almost instantaneous, typically in milliseconds.

With this in mind, Let’s look into the performance aspects of each

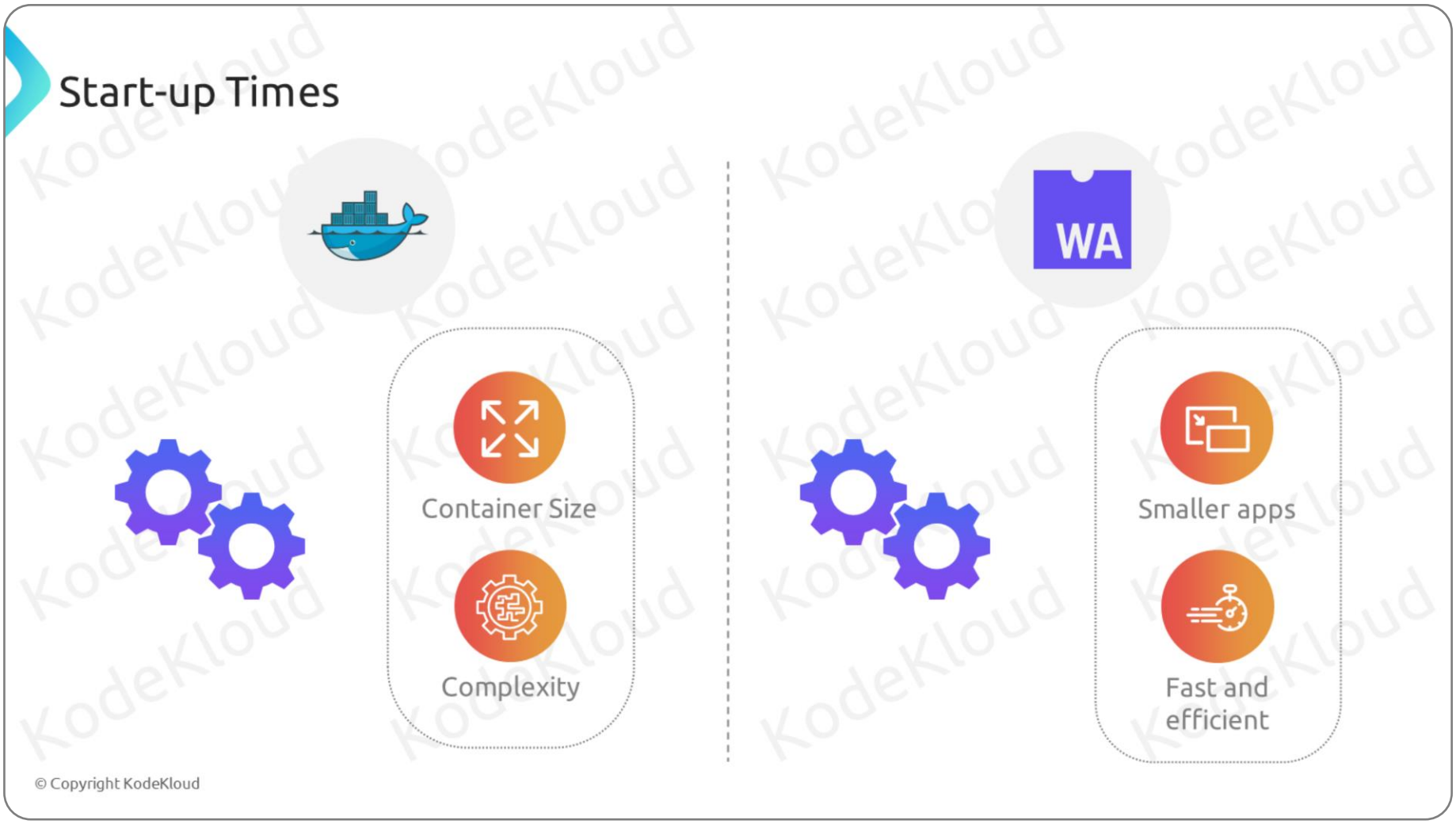
Start-up Times




```
$ docker run -p 3030:3030 rust-hello-world
```



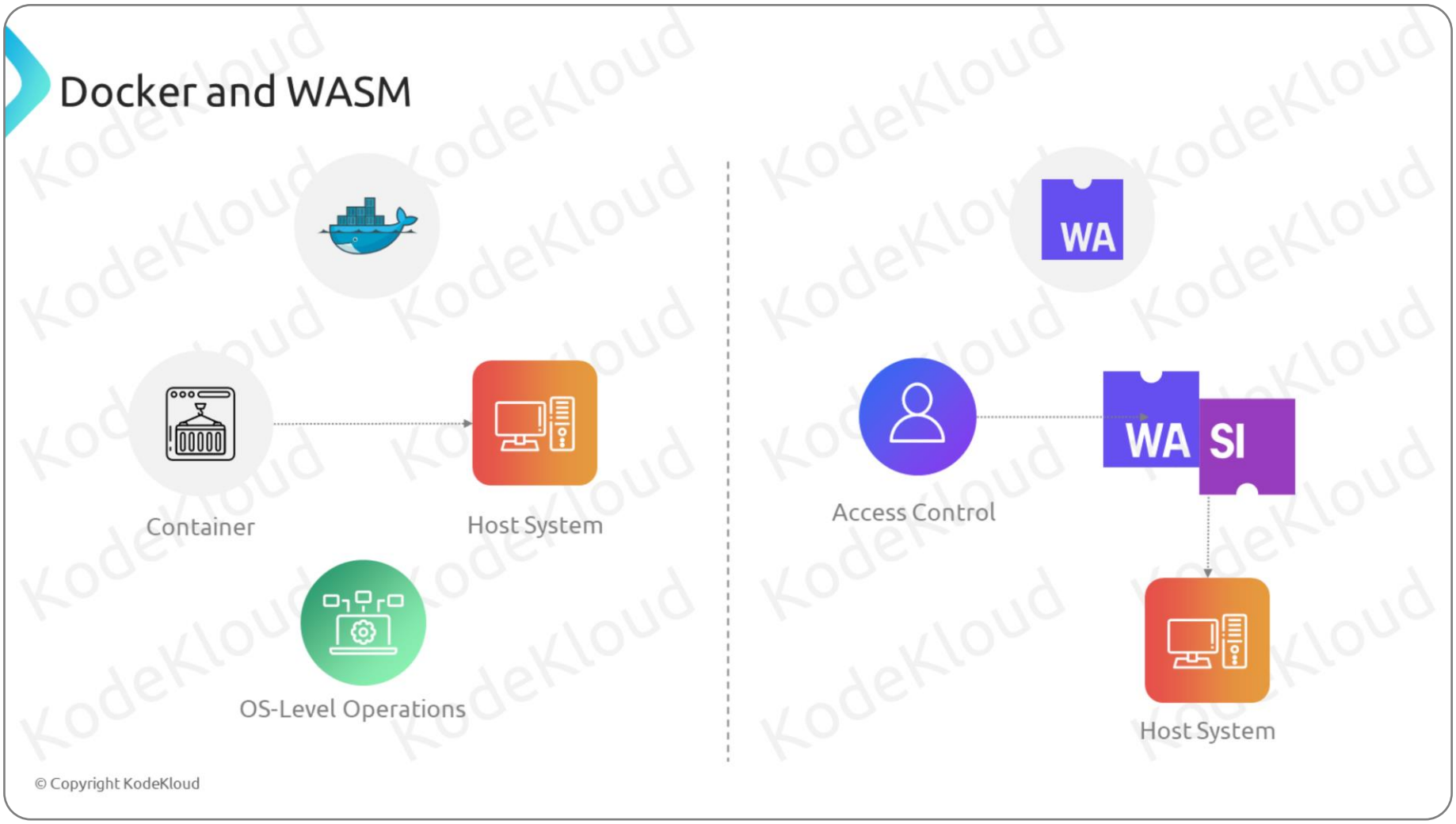

```
$ wasmtime target/wasm32-wasi/debug/myapp.wasm
```



Docker is generally pretty fast and behaves a lot like software that's directly installed on a computer. But, its speed can be affected by how big the container is and how complex the software inside it is.

WebAssembly (WASM), on the other hand, really stands out in terms of speed and efficiency. It's particularly good for smaller, more focused apps, where its design for fast and efficient performance really makes a difference

Then there's the matter of running in web browsers.
Docker wasn't really made for working inside web browsers. It's mainly used for applications that run on servers or as standalone software on different kinds of computers.
WASM, however, was created with the web in mind. It's great at running inside browsers, allowing powerful applications to work right in your web browser, something that's not really in Docker's wheelhouse.



Finally, the interaction with the Host System

Containers have a more intimate relationship with the host system. They can interact more directly with system resources and OS-level operations, given their encompassing nature.

WASM operates at a higher level of abstraction. Its interaction with the system is mediated through WASI that we have

already discussed in previous lessons, which limits its direct access to system resources. This provides security benefits but can limit its capability for certain types of system interactions.

Docker and WASM

Now that we've explored Docker and WASM individually, it's time to see how these two powerful technologies can be combined for even greater benefits. It's not about replacing one with the other, but rather about harnessing their strengths together.

WebAssembly (WASM) has carved its niche with its ability to run code written in languages like C, C++, Rust, or Go with near-native performance, high security, and rapid startup times.

WebAssembly (WASM) has carved its niche with its ability to run code written in languages like C, C++, Rust, or Go with near-native performance, high security, and rapid startup times.

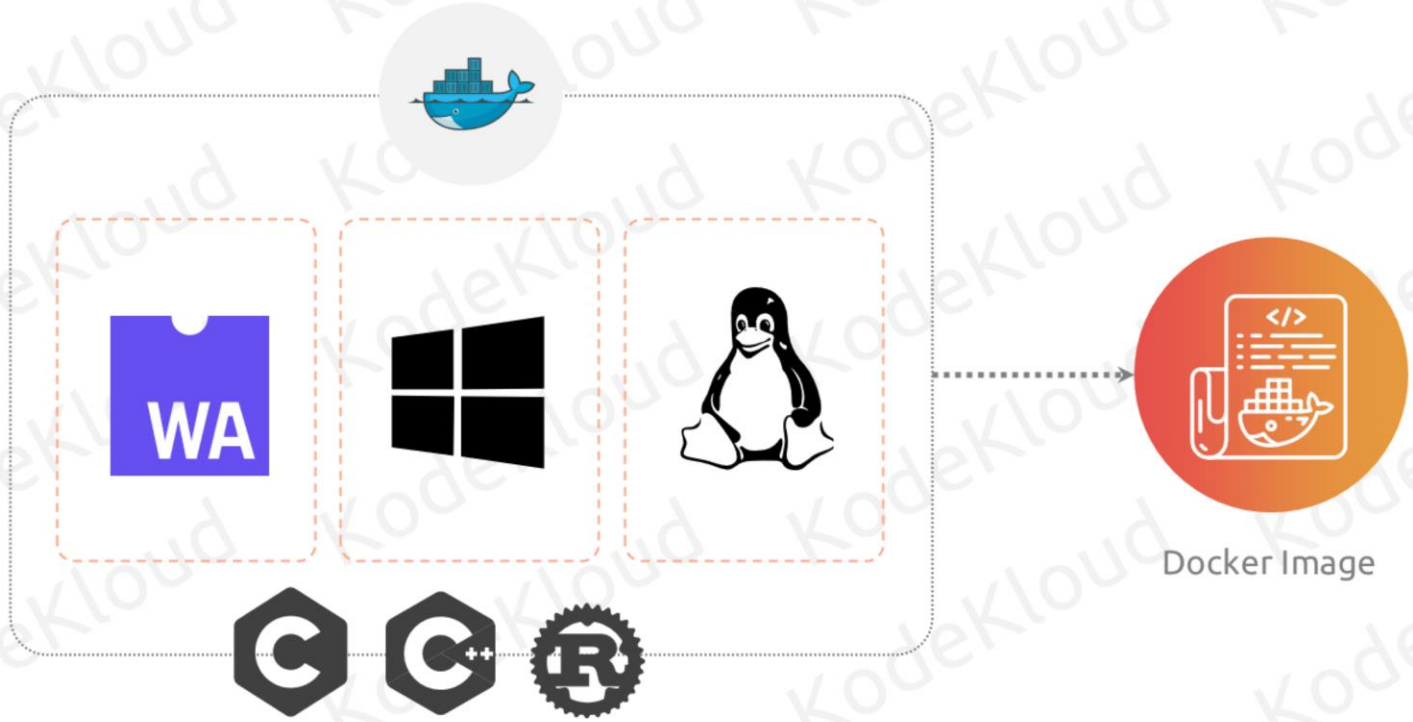
Docker, on the other hand, has revolutionized the way we package and deploy applications, offering high portability and runtime isolation. The question then arises: do we have to choose between these two technologies? The answer is a

resounding no.

Docker and WASM

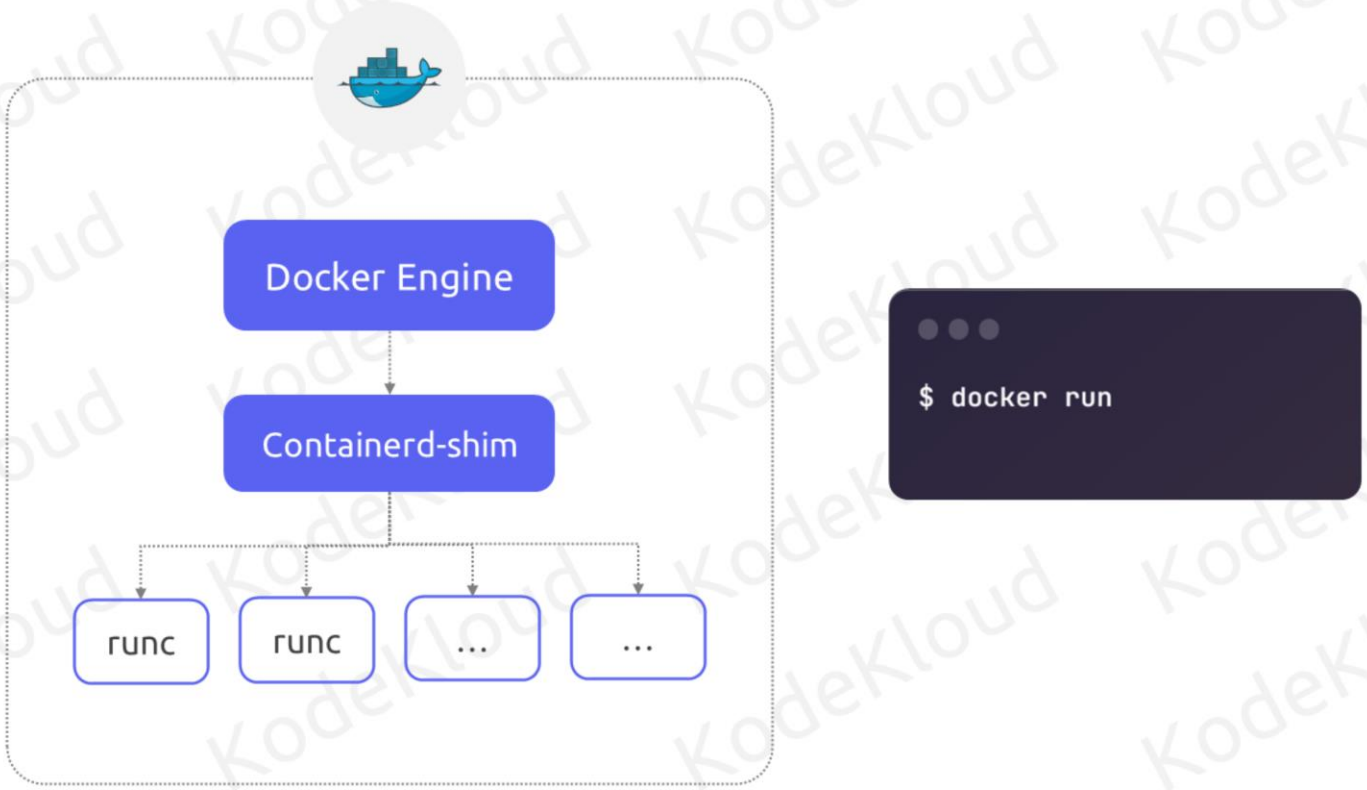
Imagine a scenario where the high performance and security of WASM meet the portability features of Docker. This isn't just a theoretical exercise but a practical integration that's already underway.

Docker and WASM



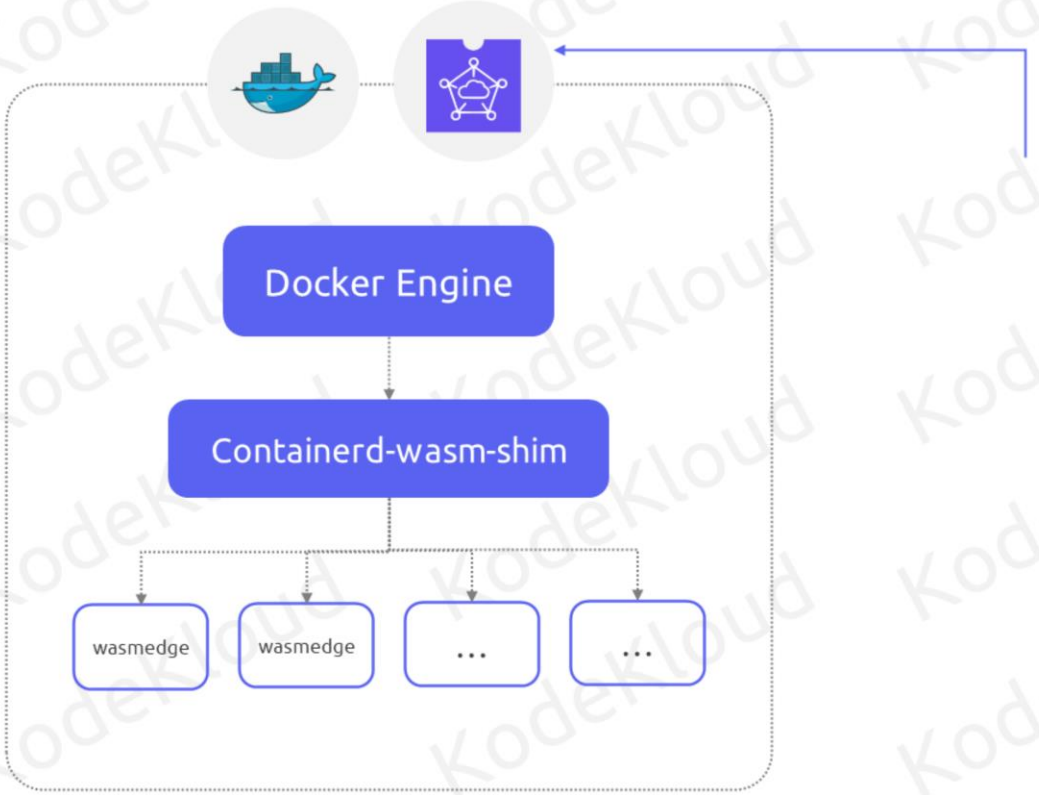
With Docker's evolving support for WASM, we're stepping into a new era of software deployment. This integration allows WASM containers to run alongside traditional Linux and Windows containers in Docker. This means you can package your native application code (written in languages supported by WASM) within a WASM container and then share it as a Docker image.

Docker and WASM



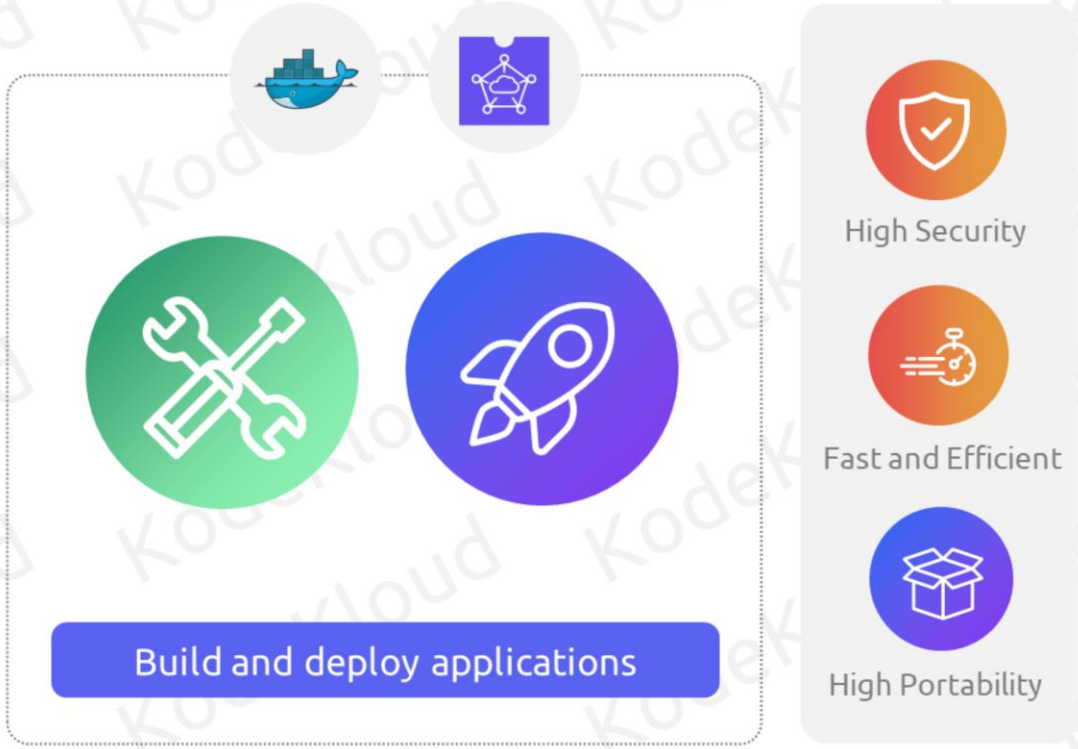
Let’s have a look at how docker and wasm work together.
At Docker's core is the Docker Engine, which processes all your container-related requests. When you run a container with the docker run command, the Docker Engine uses a runtime like runc to start the container process. Post this initiation, a containerd-shim manages the running container, overseeing aspects like log capturing and process management.

Docker and WASM



For WASM containers, Docker uses a special runtime called wasmedge. To integrate this with Docker, a new containerd-shim, named containerd-wasm-shim, has been developed. This shim ensures that WASM containers can be managed just like any other Docker container, making the process seamless.

Benefits to Developers



What does this mean for developers and IT professionals?
You can now build and deploy applications that combine the efficiency and performance of WASM with the universality and ease of Docker. Whether it's running a microservice, executing a function, or deploying a full-fledged application, this integration broadens the scope of what can be achieved with containerization.
wrapping up with how devs can benefit from this.

Benefits to Developers

In it's early stage

Available in Docker Desktop

Enhanced implementation and Management

It's important to note that this exciting integration is still in its early stages, available in a special technical preview build of Docker Desktop. But the implications are vast. As the integration matures, it could significantly enhance how applications are developed, deployed, and managed, especially in cloud-native environments like Kubernetes clusters.

Let's take a look at wasm and docker integration in the coming demo..
heads up on the demo related to how wasm and docker works together. - specifically, to run a c progam inside dcoekrr

container.

WASM Tools and Frameworks

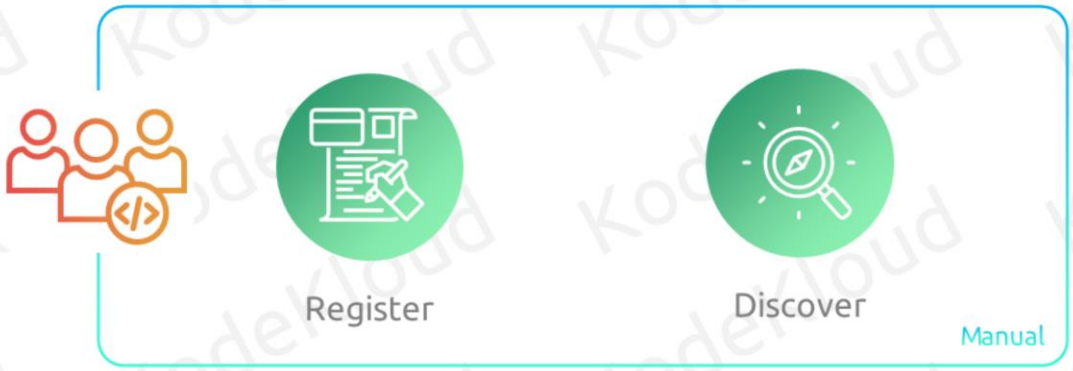
WASM Application – Service Discovery and Load Balancing

Service Discovery

Load Balancing

In distributed applications built with WebAssembly, service discovery and load balancing become even more critical.

WASM Application – Service Discovery and Load Balancing



Without a dedicated framework, WebAssembly developers find themselves in a situation where they need to manually register and discover services. For instance:

WASM Application – Service Discovery and Load Balancing

```
// Manually register a WebAssembly service
registerWasmService("serviceA", "https://localhost:8080");

// Manually discover a WebAssembly service
serviceEndpoint = discoverWasmService("serviceB");

// Manually handle load balancing for WebAssembly services
selectOptimalEndpoint(serviceEndpoint);
```

Register

Discover

Load Balancing

Manual Implementation

Error Prone

Such manual implementations can be error-prone and lack the sophistication of dedicated solutions that handle these tasks efficiently.

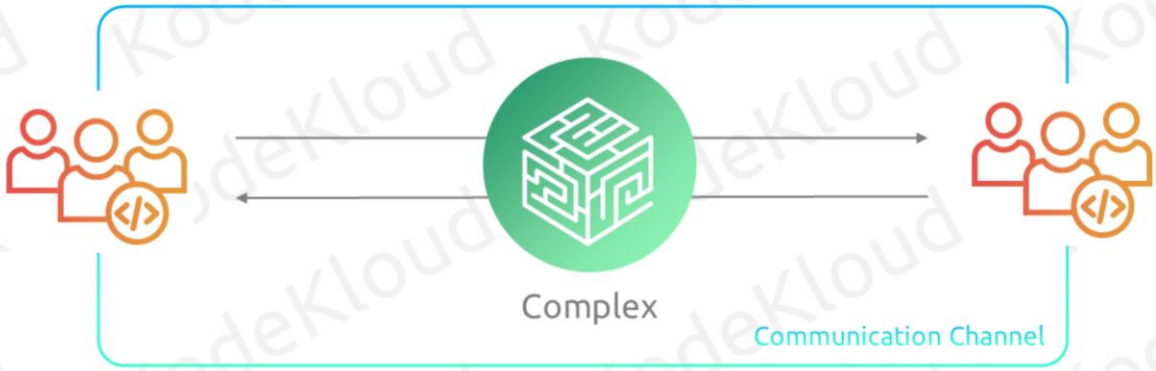
WebAssembly Application – Inter-Service Communication

Effective Communication

Next we should look into Inter-service Communication in a WebAssembly Application.

Effective communication between WebAssembly services is the backbone of any distributed system. Without a framework, setting up these communication channels can

WebAssembly Application – Inter-Service Communication



Effective communication between WebAssembly services is the backbone of any distributed system. Without a framework, setting up these communication channels can become complex. Consider the following:

WebAssembly Application – Inter-Service Communication

```
// Manually set up communication between WebAssembly services
setupWasmCommunication("serviceA", "serviceB");

// Send a message from one Webassembly service to another
sendMessage("serviceA", "serviceB", message);
```

Here, there's no standardized way to handle message formats or communication protocols, leading to potential

WASM Application – Security

Security

Then, Let’s consider the security in an application developed using WebAssembly.

WASM Application – Security

Security is paramount, especially in distributed systems built with WebAssembly. However, without a dedicated framework, developers might find themselves implementing security features like authentication and encryption manually:

WASM Application – Security

```
// Manually implement authentication for WebAssembly services
authenticateWasmUser(username, password);

// Manually handle data encryption for WebAssembly data
encryptedData = encrypt(data, encryptionKey);
```

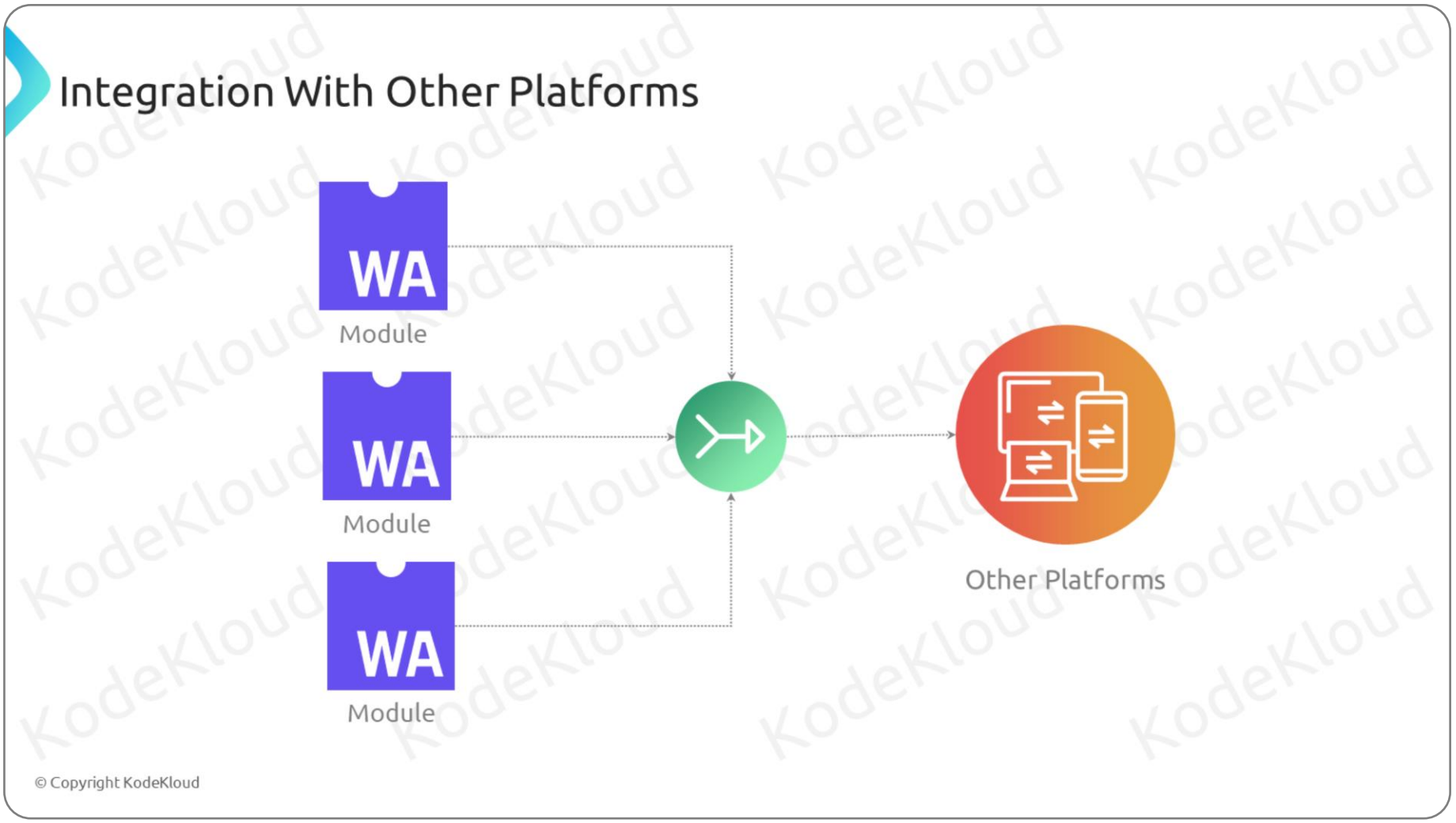
Authentication

Encryption

Manual Implementation

Vulnerability

Such manual implementations can introduce vulnerabilities if not done correctly and may not keep up with the latest security best practices.



Next, WebAssembly modules often need to integrate with various platforms. Without a framework, this integration can become cumbersome, leading to platform-specific code:

Integration With Other Platforms

```
// Manually integrate a WebAssembly module with a specific platform
platformIntegration("platformA", wasmModule);

// Handle platform-specific quirks for WebAssembly modules
if (platform == "platformA") {
  // PlatformA-specific WebAssembly code
} else if (platform == "platformB") {
  // PlatformB-specific WebAssembly code
}
```

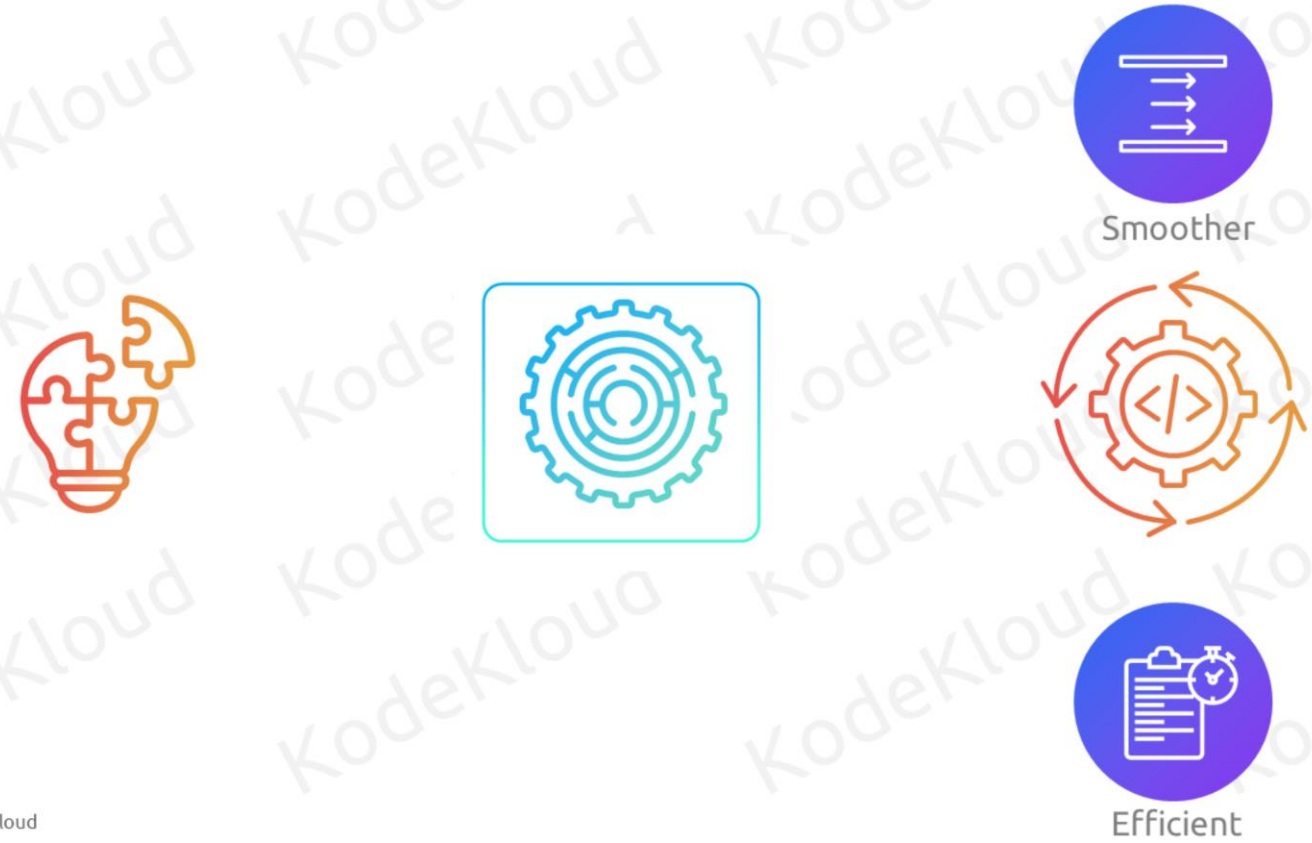

Such code can lead to tight coupling, making it hard to adapt the WebAssembly application to new platforms or changes in existing platforms.

Extensive setup code

Boilerplate code can make a WebAssembly application bloated and harder to maintain. Without a framework, developers might find themselves writing extensive setup code:

Such tightly coupled code reduces modularity, making it challenging to modify or extend the WebAssembly application.

WASM Frameworks



Now, imagine if there were solutions tailored to address these very challenges, making the development process smoother and more efficient. Well, that's where our focus will be in this lesson.

WASM Frameworks

WASM Frameworks

WasmCloud

Fermyon Spin

Transitioning to our main topic, we have two powerful solutions that have been designed to bridge the gaps in the Wasm ecosystem: WasmCloud and the Fermyon Spin framework.

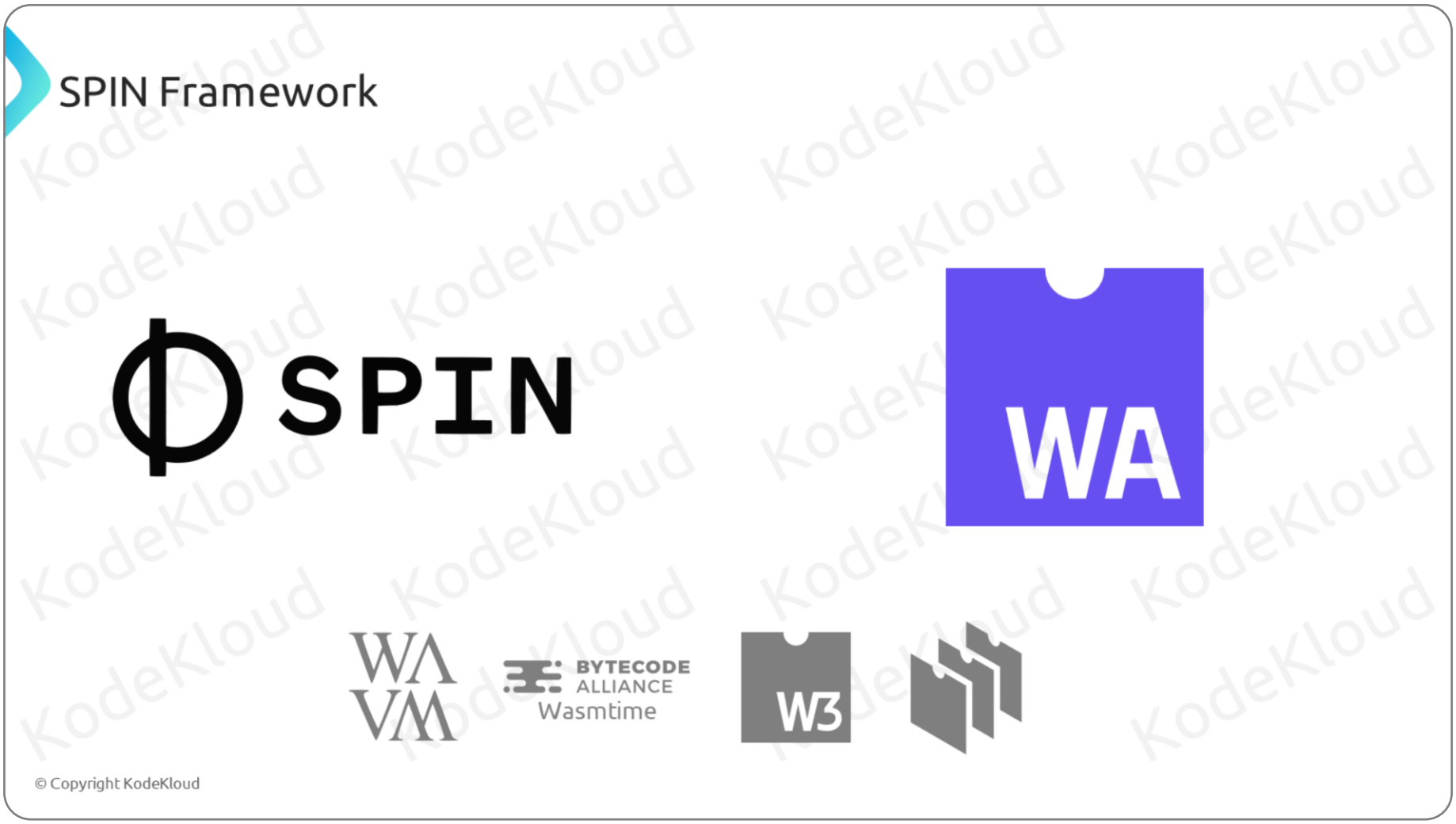
SPIN Framework

Let’s start with Spin.

Spin is an open-source framework designed specifically for WebAssembly. Its primary focus is on building, deploying, and running cloud microservices that are fast, secure, and composable.

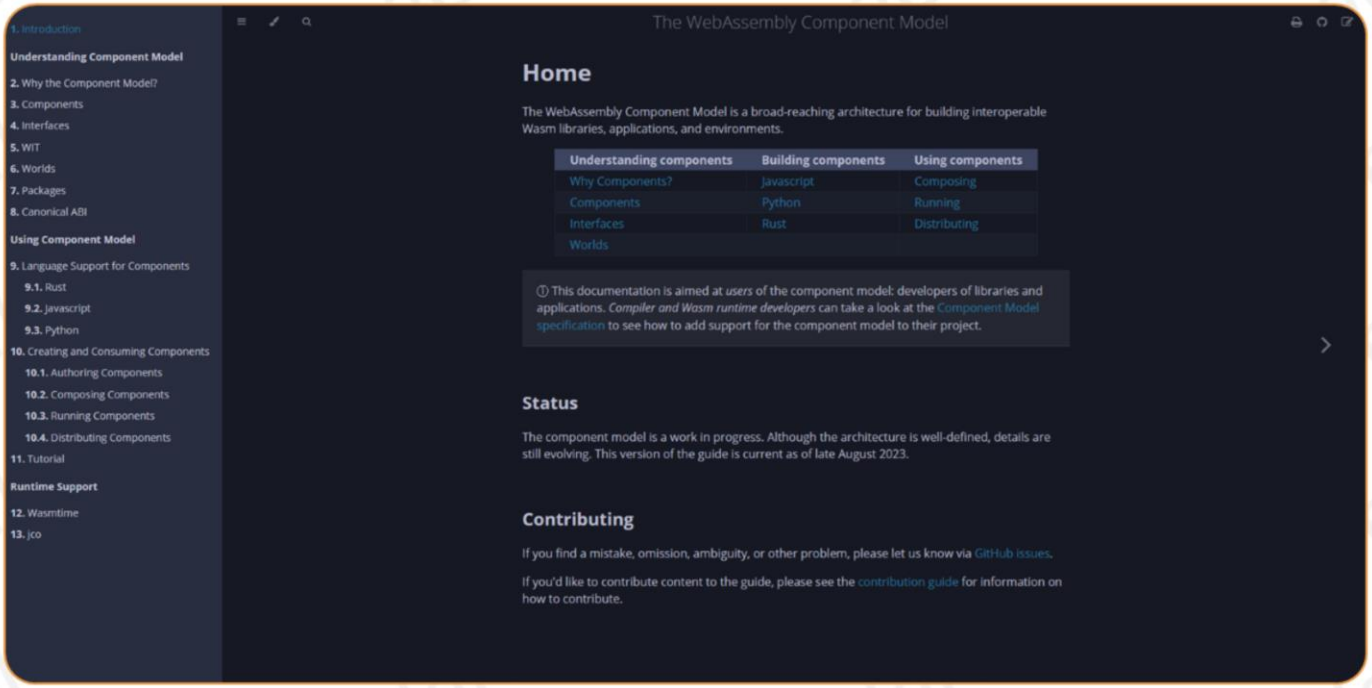
SPIN Framework

SPIN

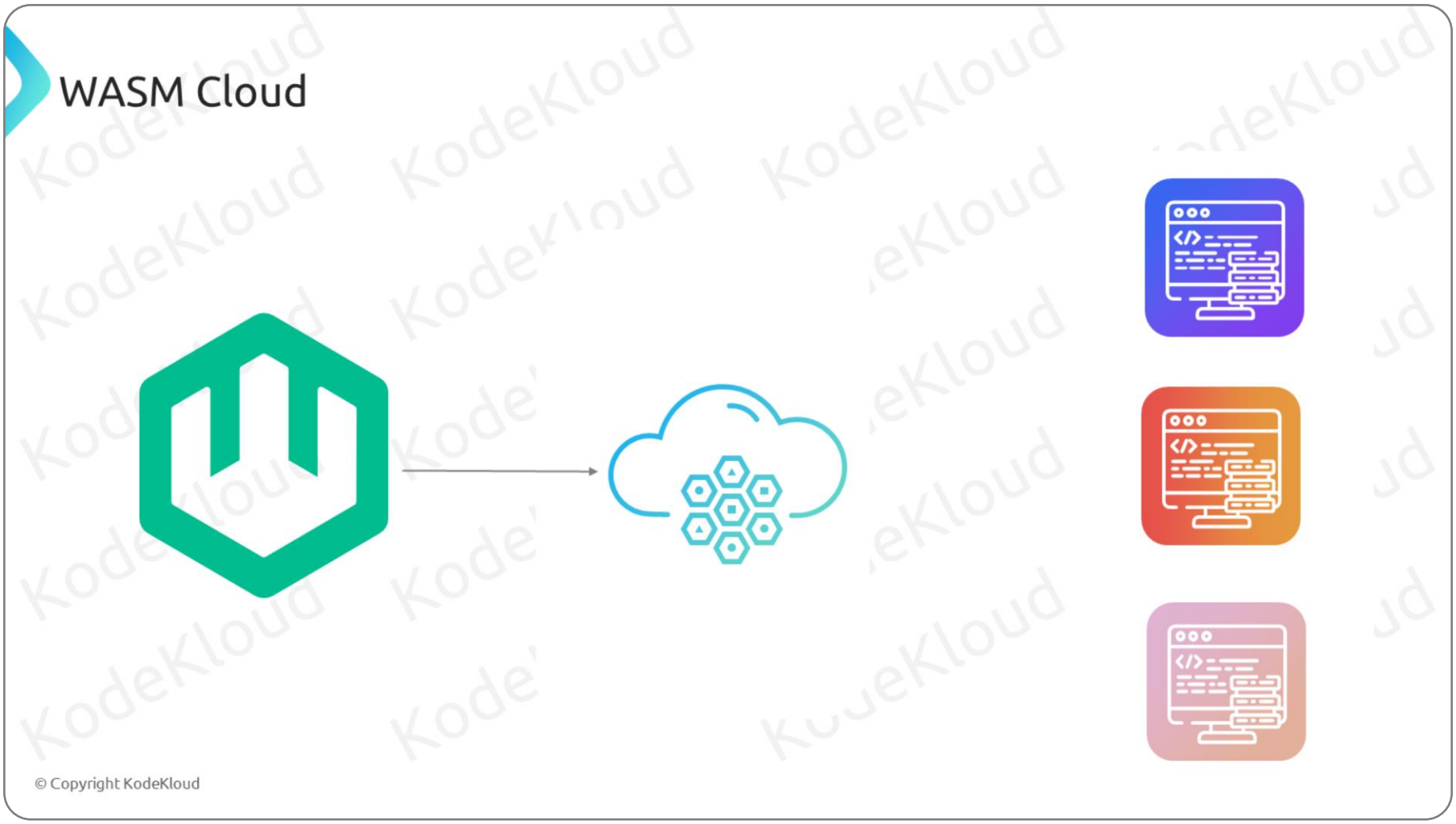


One of the standout features of Spin is its alignment with the latest advancements in the WebAssembly component model, coupled with the Wasmtime runtime.

SPIN Framework



https://component-model.bytecodealliance.org/

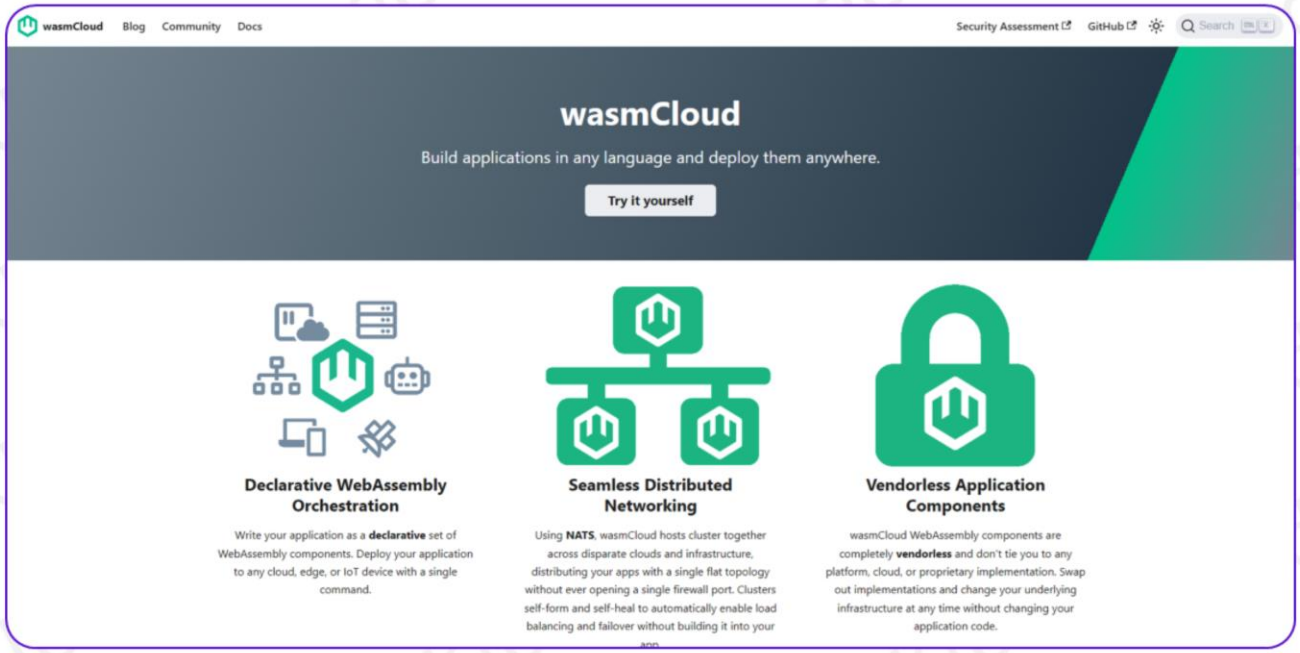


Next, Wasmcloud.

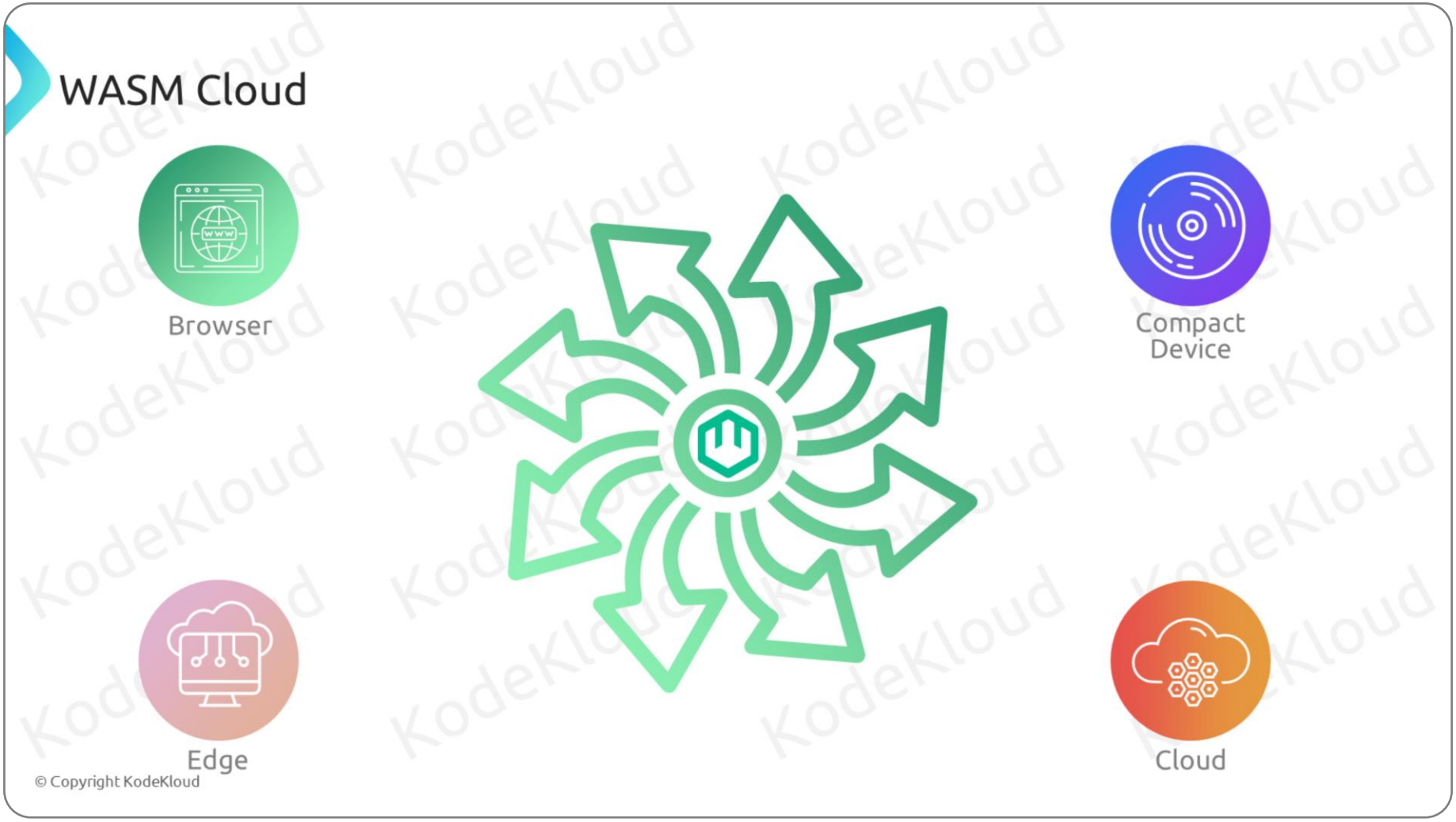
wasmcloud positions itself as a game-changer in the domain of distributed computing. It serves as a versatile runtime, capable of running applications not just in the cloud but also at the edge, within browsers, on compact devices, and virtually any conceivable environment.

It promises to retain all the robust features expected of enterprise-grade systems. With wasmCloud, the emphasis is on productivity.

WASM Cloud



https://wasmcloud.com/



Next, Wasmcloud.

wasmcloud positions itself as a game-changer in the domain of distributed computing. It serves as a versatile runtime, capable of running applications not just in the cloud but also at the edge, within browsers, on compact devices, and virtually any conceivable environment.

WASM Cloud

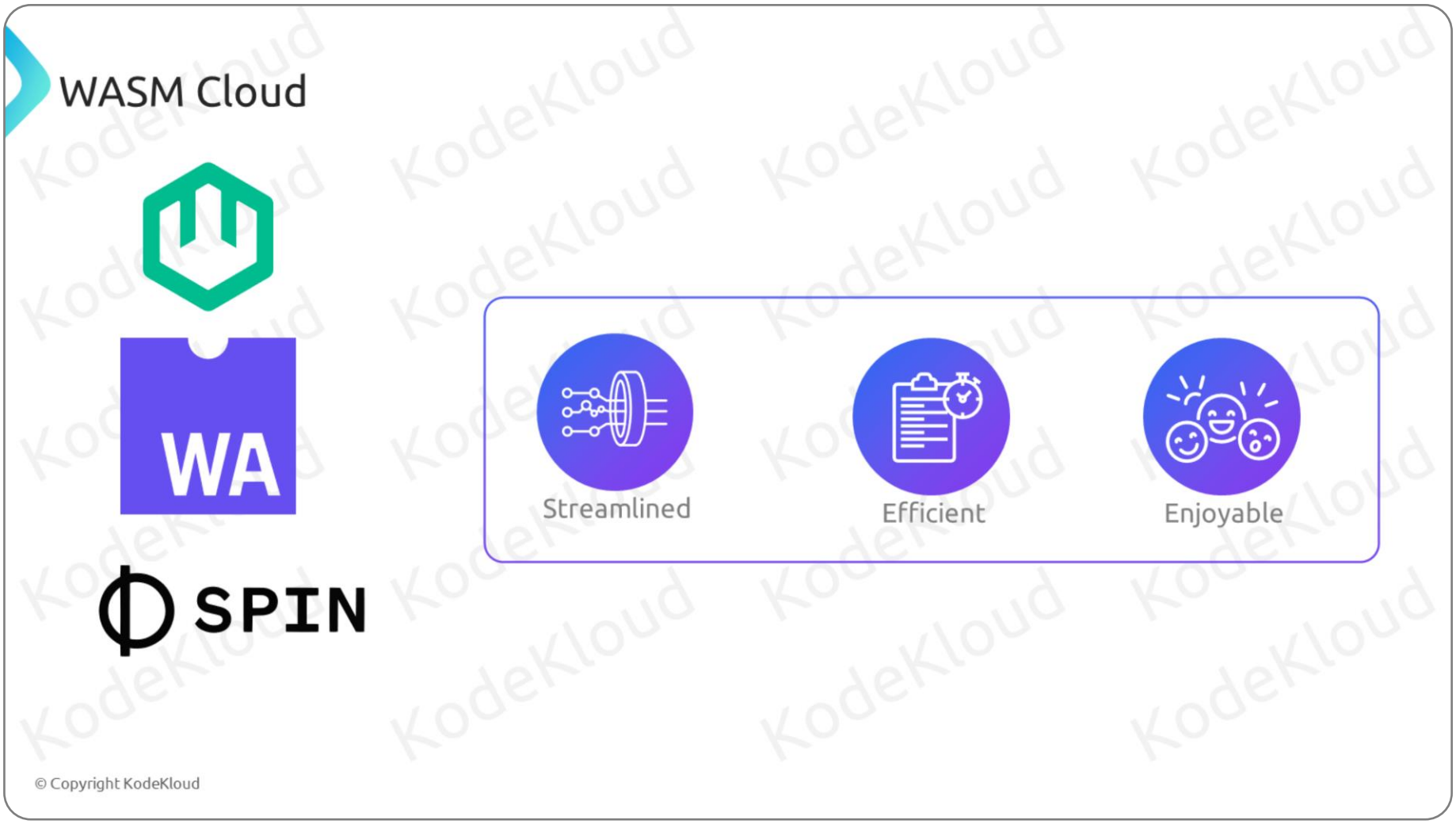
It promises to retain all the robust features expected of enterprise-grade systems.

With wasmCloud, the emphasis is on productivity.

WASM Cloud

Developers can channel their efforts into delivering features without getting bogged down by common pain points like excessive boilerplate code, dependency issues, tight coupling, or restrictive infrastructure designs.

WASM Cloud



By leveraging tools like wasmCloud and Spin, you can significantly enhance your WebAssembly application development process. These tools aim to address the gaps we discussed, providing you with a more streamlined, efficient, and enjoyable development experience.

WASM Cloud

WASM Cloud

So, gear up, explore these tools, and embark on a transformative WebAssembly development journey.

This forward-looking section explores the role of WebAssembly in cloud computing. It discusses how WebAssembly extends beyond browser applications into cloud environments, detailing WebAssembly runtimes and their integration with cloud-native technologies. The section looks at WebAssembly's interaction with Docker, showcasing practical integration scenarios. It also introduces key tools and frameworks relevant to WebAssembly in cloud contexts, like WasmCloud and the Fermyon Spin framework.

Follow us on https://kodekloud.com/ to learn more about us.