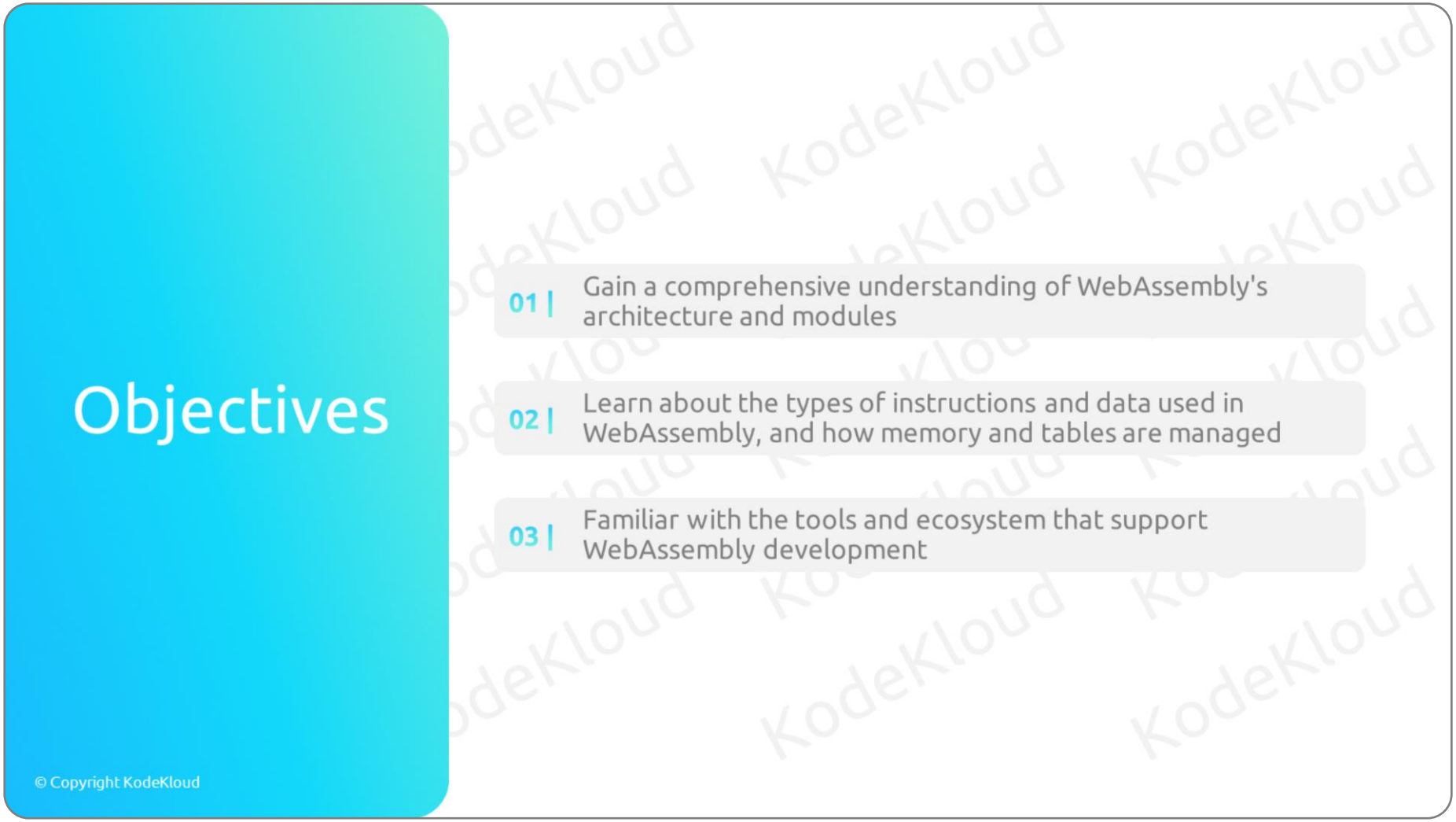
KodeKloud

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

WebAssembly Core Concepts



Learners will gain a comprehensive understanding of WebAssembly's architecture and modules. They will learn about the types of instructions and data used in WebAssembly, and how memory and tables are managed. Additionally, they will become familiar with the tools and ecosystem that support WebAssembly development.

WebAssembly Architecture

WebAssembly is changing how we build web apps.

WebAssembly

It's like a supercharged engine for the web, making things run faster and smoother. Before we dive into how it does this magic, let's understand its building blocks and how they fit together.

Binary Format

WebAssembly's binary format is a compact representation of code that's optimized for speed.

Binary Format

Unlike human-readable text, this format is designed for machines, ensuring quick decoding and execution in browsers.

Binary Format

C Program

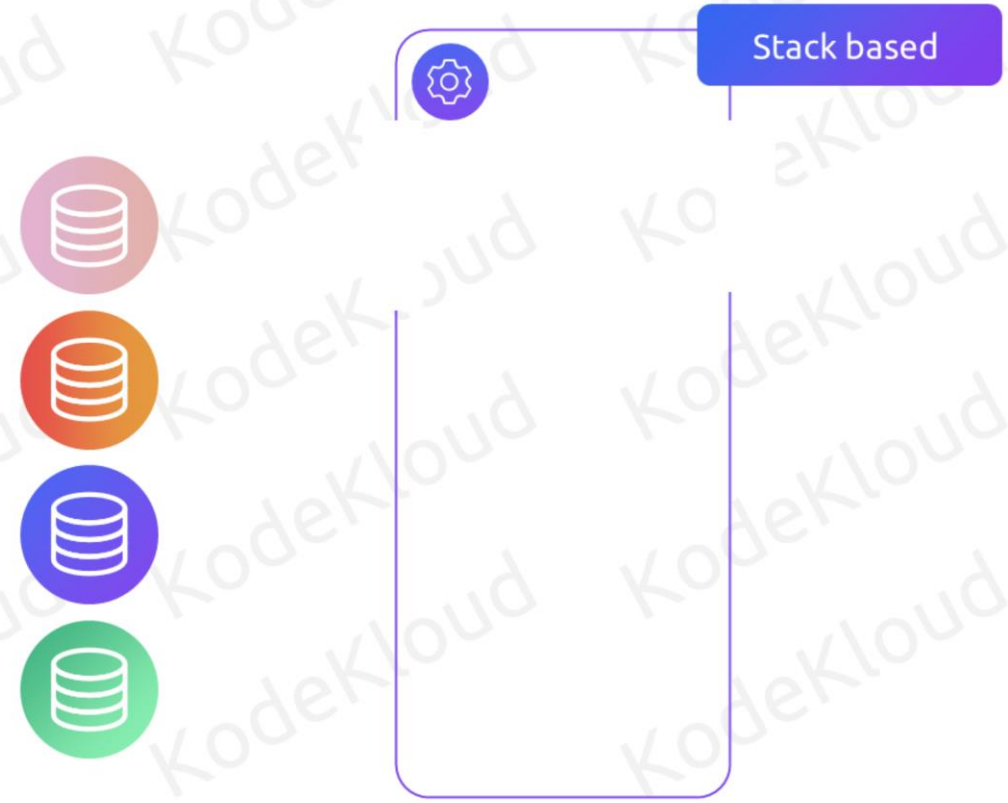
Example:
Consider a simple function in C:
// code //

Binary Format

```
int add(int a, int b) {
    return a + b;
}
```

When this function is compiled to WebAssembly, it's transformed into a series of bytes in the binary format. This binary is much smaller and faster for browsers to read and execute than the original C code.

Stack-Based Virtual Machine



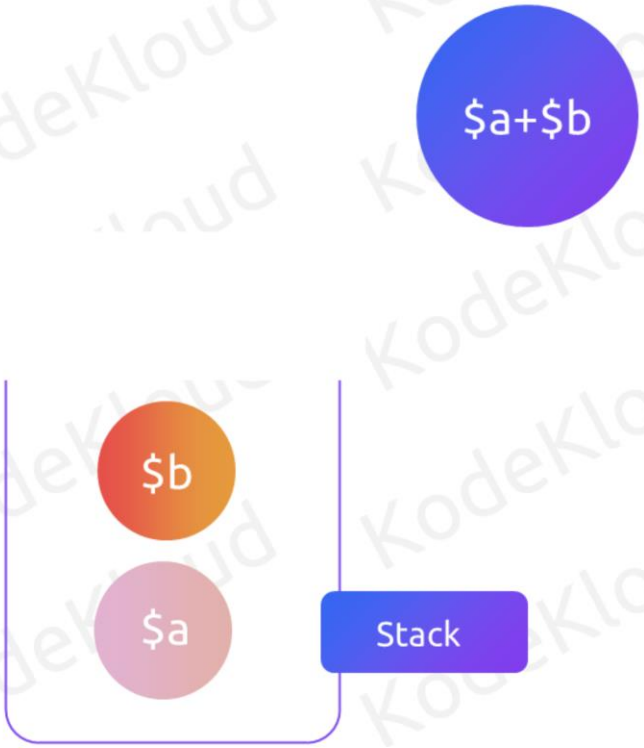
WebAssembly operates on a stack-based virtual machine. This means it uses a last-in first-out (LIFO) approach where values are pushed onto a stack, operations are performed, and results are popped off the stack.

Stack-Based Virtual Machine

Last-in First-out
(LIFO)

Stack-Based Virtual Machine

```
(func $add (param $a i32) (param $b i32) (result i32)
    get_local $a
    get_local $b
    i32.add
)
```

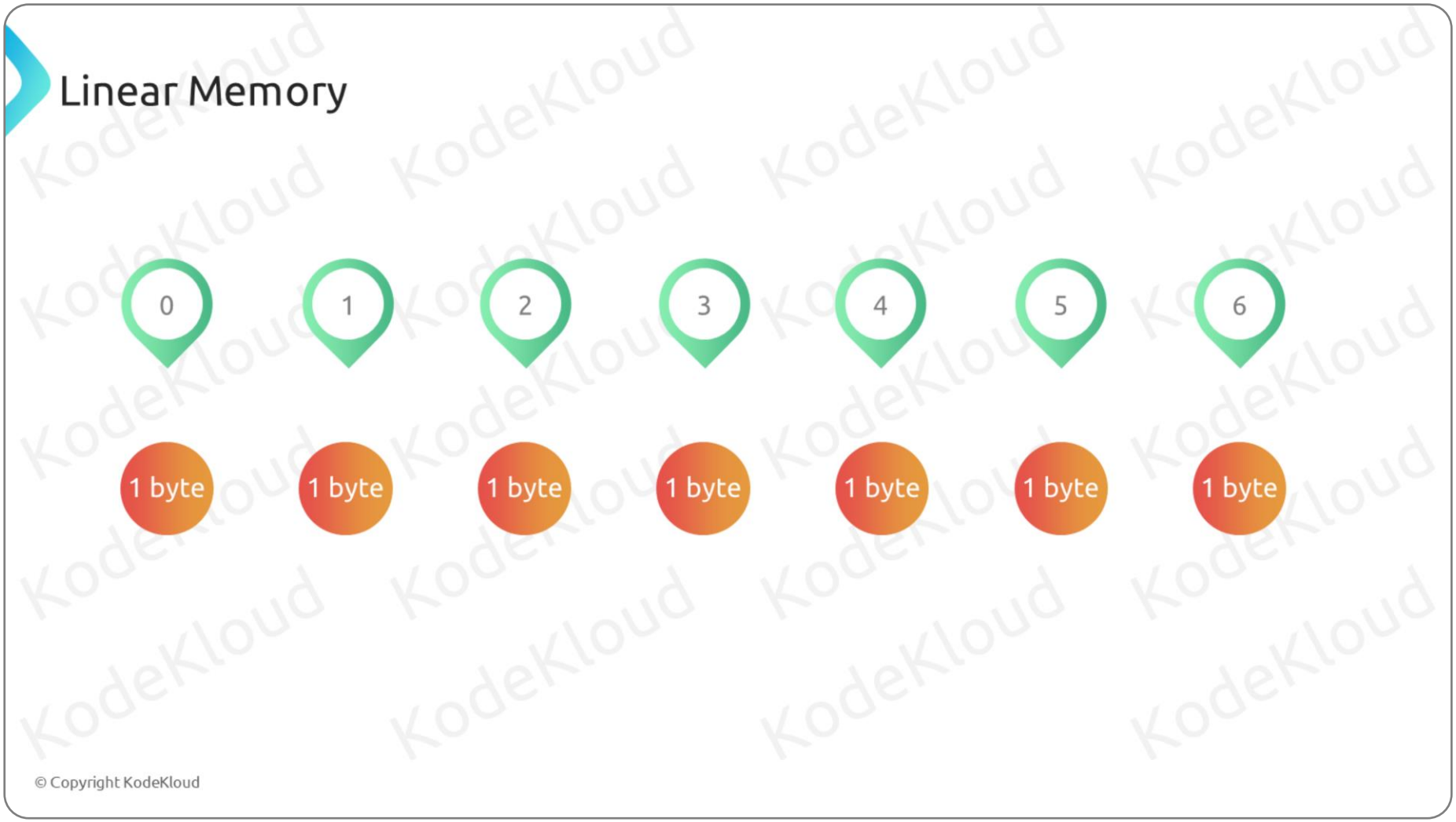


Example:
The addition of two numbers in WebAssembly's text format might look like:

// code //

This function takes two integers, pushes them onto the stack, performs the addition, and then pushes the result back onto

the stack.

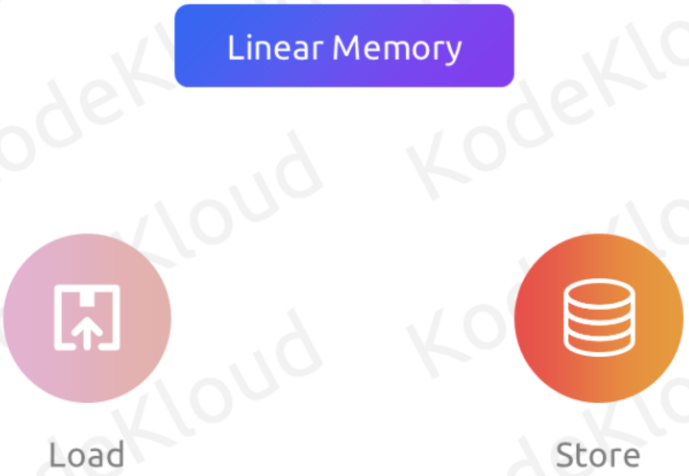


WebAssembly's linear memory is like a big array of bytes. Each byte has its own address, starting from 0 and going up.

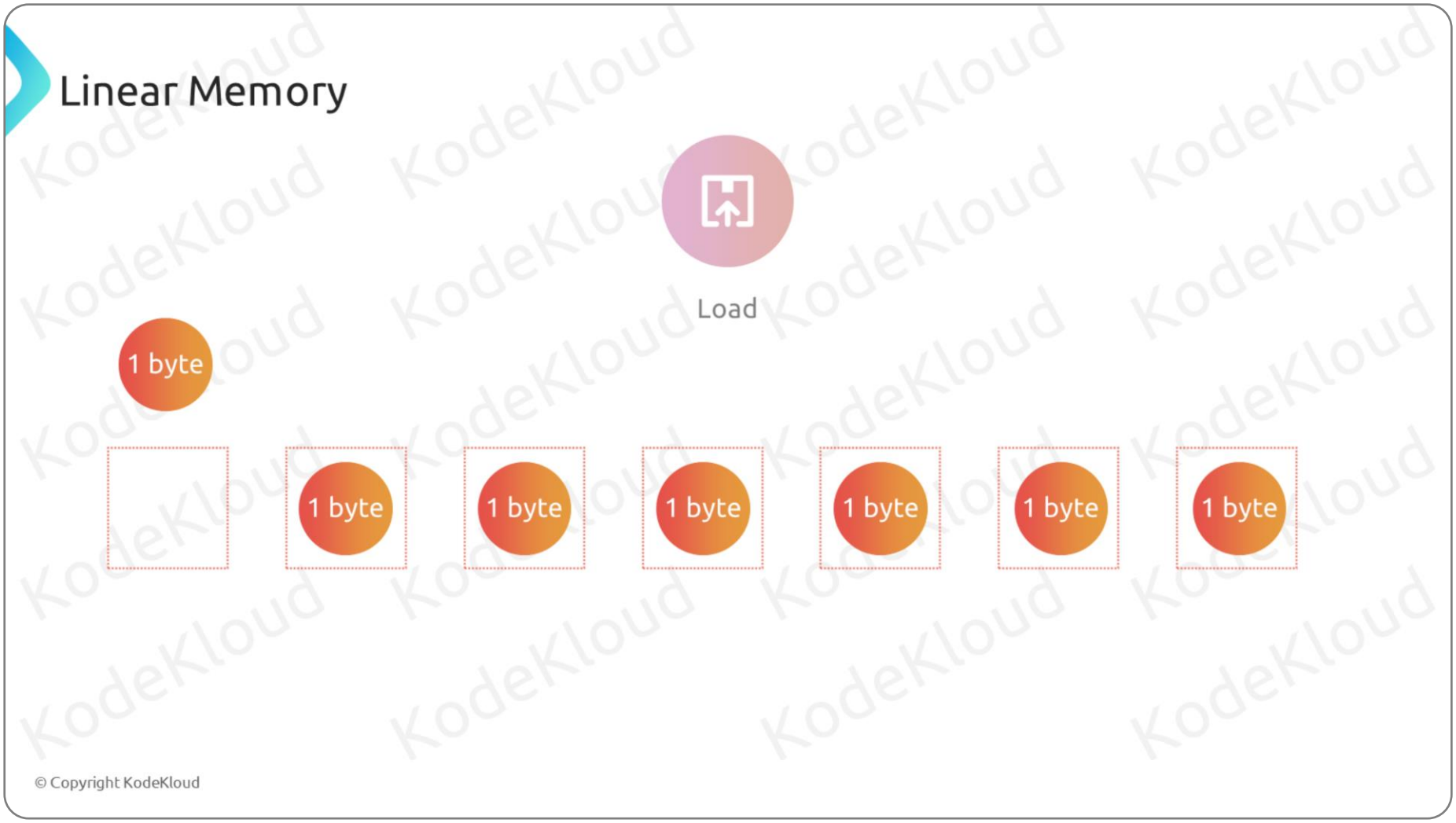
Linear Memory

This is like a long line of storage boxes, each box holding a single byte.

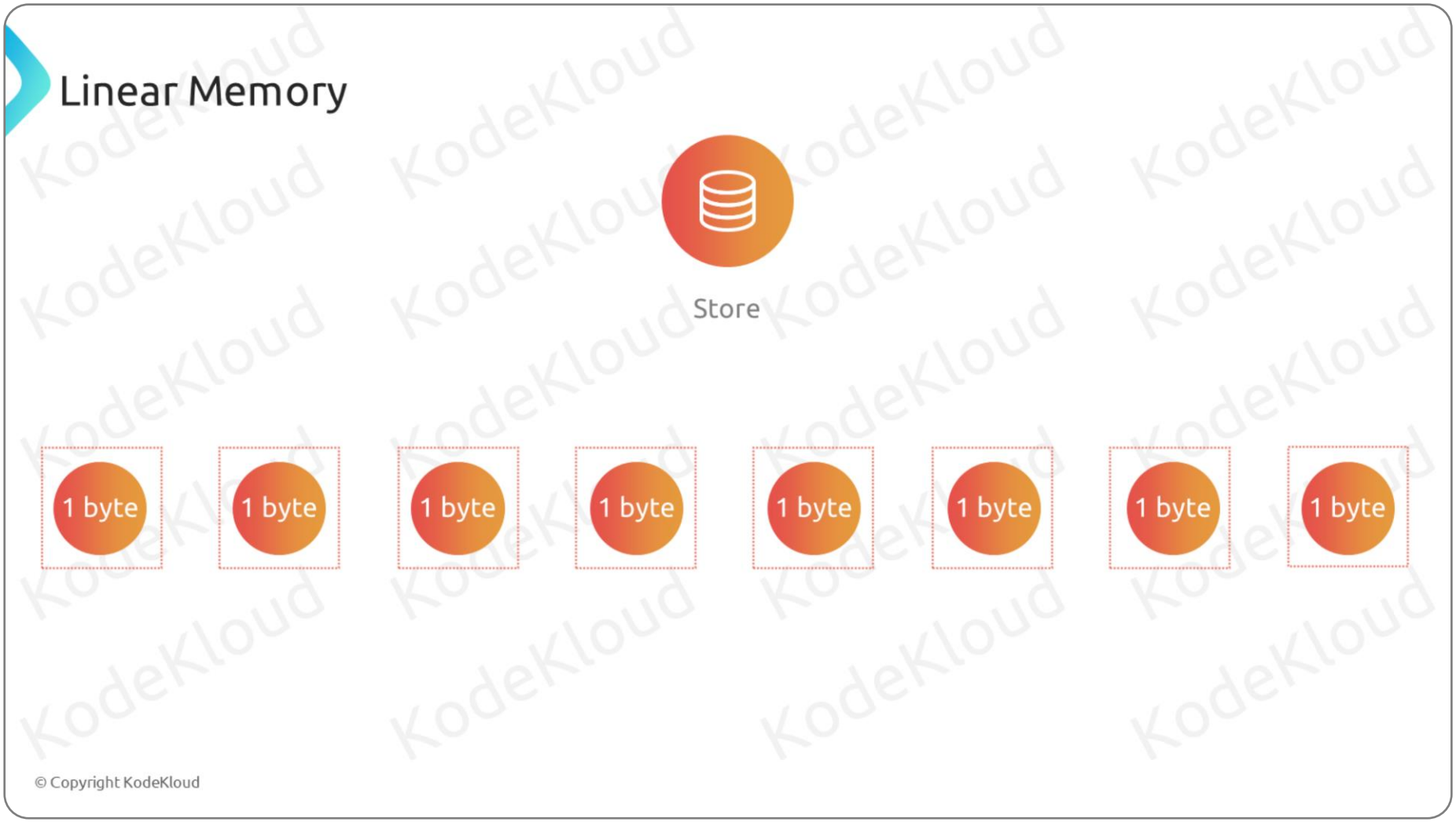
Linear Memory



Memory in WebAssembly is accessed via two primary instructions: load and store.



Load: This is like picking up something from one of the storage boxes.

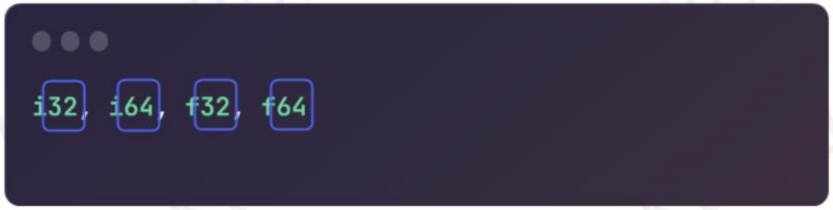


Store: This is like putting something into one of the storage boxes.

Linear Memory

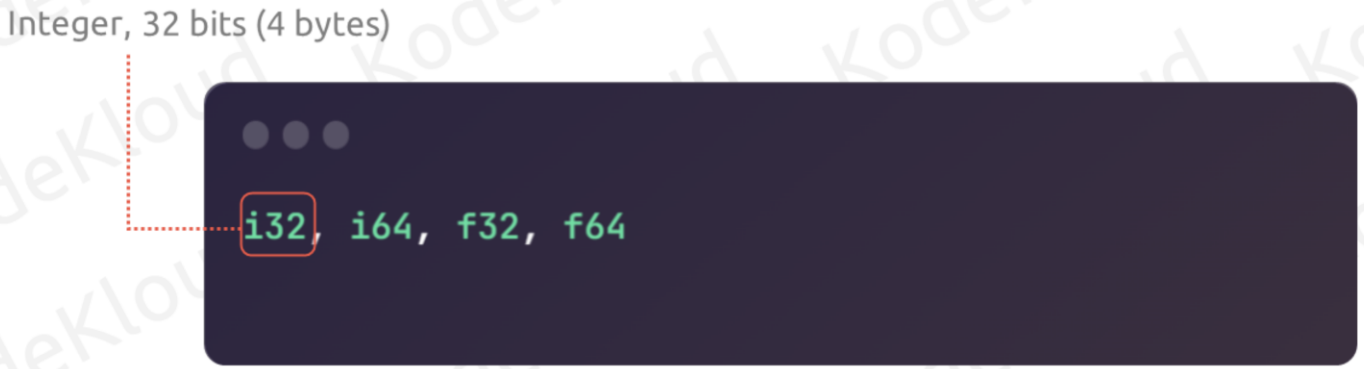
These instructions are suffixed by the type of data they handle (e.g., i32, i64, f32, f64) and the size of the data in bits when applicable

Linear Memory

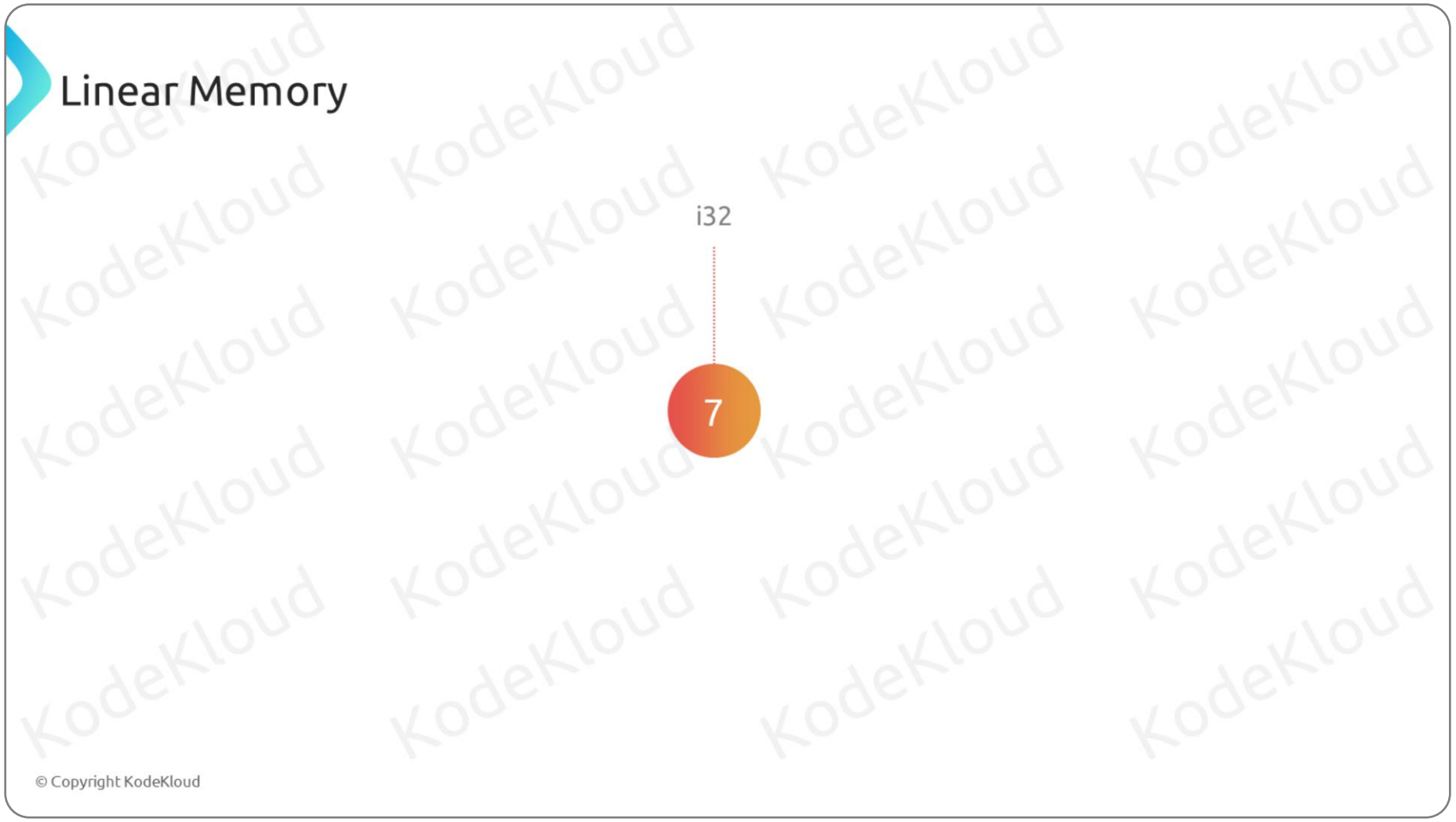


In WebAssembly, when we talk about data types like i32, i64, f32, f64, these are just ways to describe the kind of data and how much space it takes up.

Linear Memory

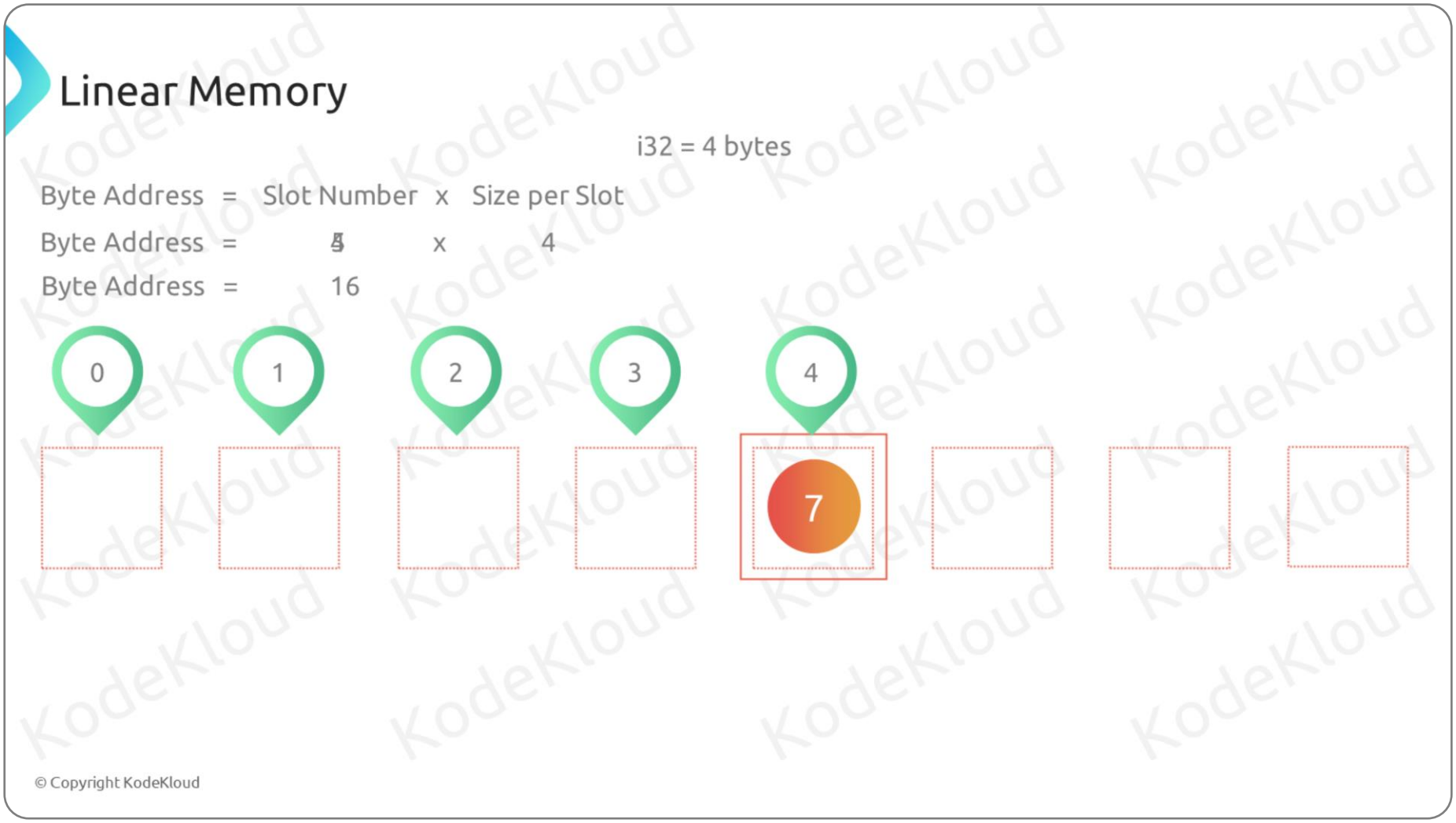


For example, i32 means an integer that takes up 32 bits (or 4 bytes) of space.



Practical Example: Storing and Retrieving a Value

Let's say you have a number, like 7, and you want to store it in this memory. We'll use our i32 type because it's a simple integer.



When it comes to Storing this data:

You choose a place to store it. Let's say you pick the fifth 32-bit slot(just for the sake of this example). To find the right byte address for this slot, you multiply the slot number by the size of each slot.
Since each i32 slot is 4 bytes, and you want the fifth slot, you multiply 4 (bytes per slot) by 5 (slot number). But, remember, we start counting from 0, so it's actually the 4th slot, and you multiply 4 by 4, which gives you 16. So, you store your number

(7) at byte address 16.

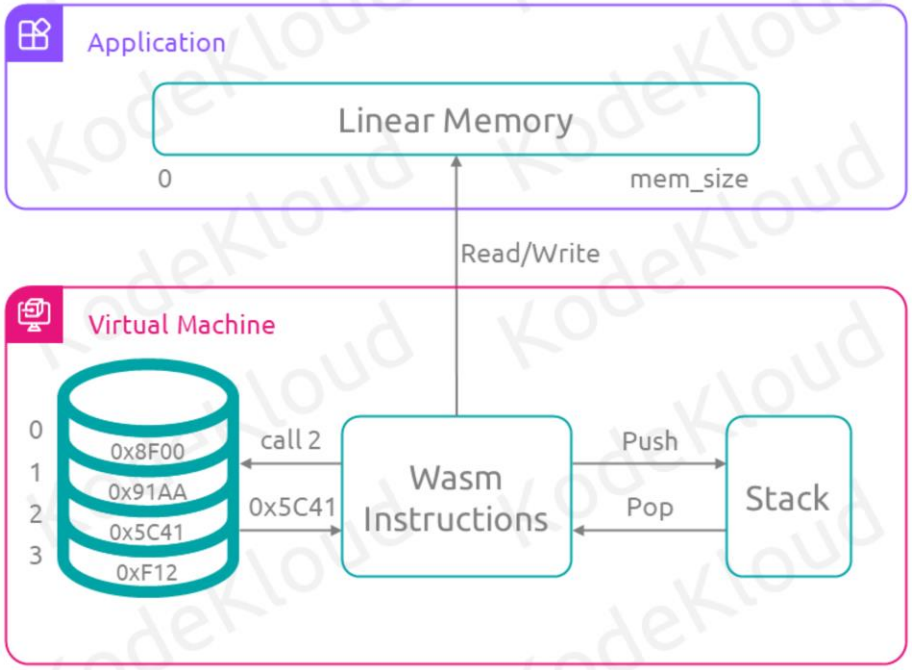
Linear Memory

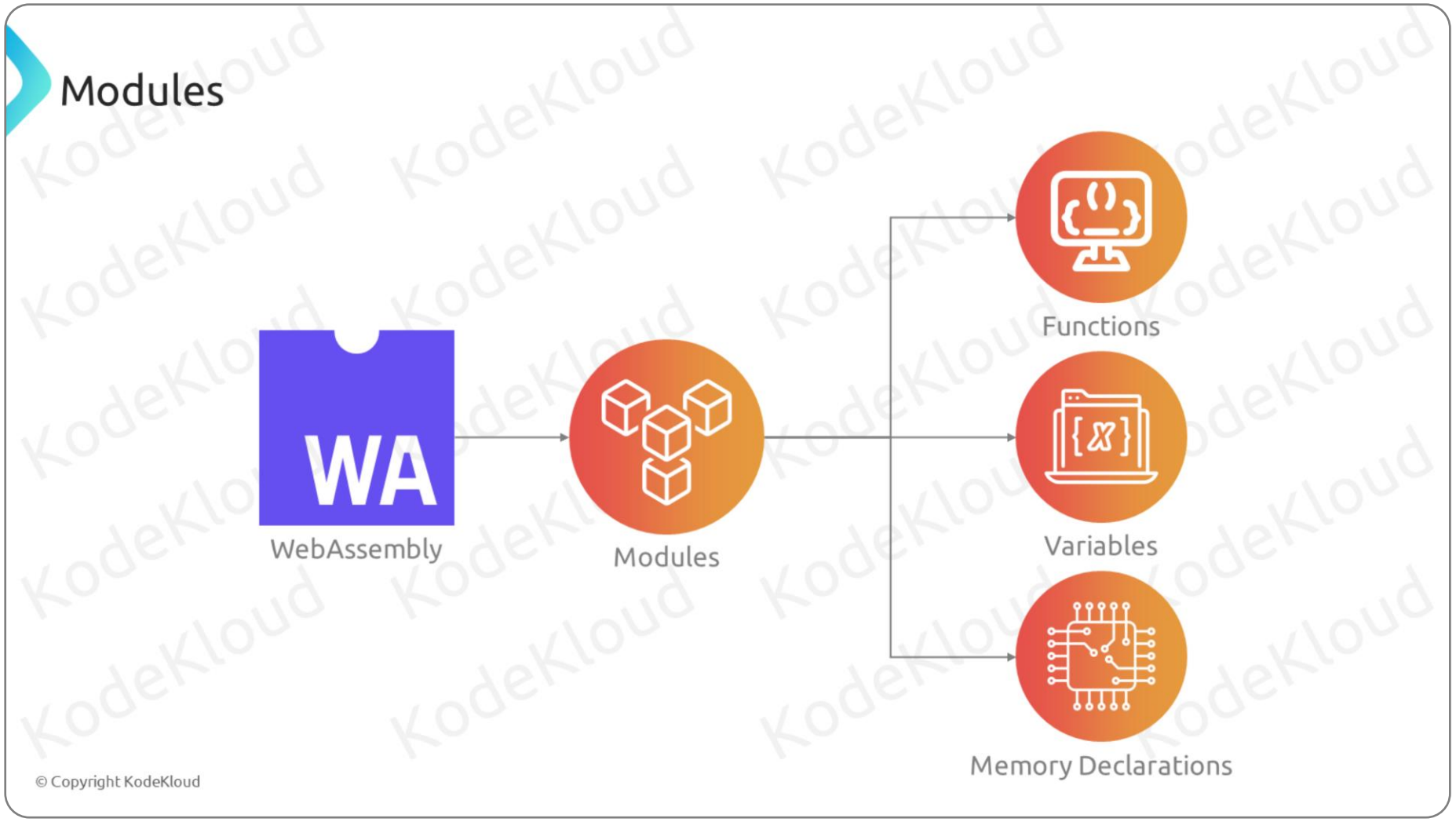
This part of the code defines a WebAssembly module with a function to store the value 7 at byte address 16. Here, we have used the WebAssembly Text Format(WAT). We start by declaring a block of memory for the WebAssembly module. Think of it like setting up a big shelf for storage. Next, We create a function named storeValue. This function's job is to put the number 7 into a specific spot in our memory shelf. Then, We choose to place the number 7 at the 16th byte position on the shelf (memory). Finally, We make this storeValue function available to be used outside of this WebAssembly module, like from a webpage's JavaScript.

Let’s have a look at the Retrieving part.

To get this number back, you just go to the same address (byte 16) and read the 4 bytes from there as an i32 value.
We create another function named loadValue. This function's purpose is to look at the 16th byte position on our memory shelf and retrieve whatever is stored there.

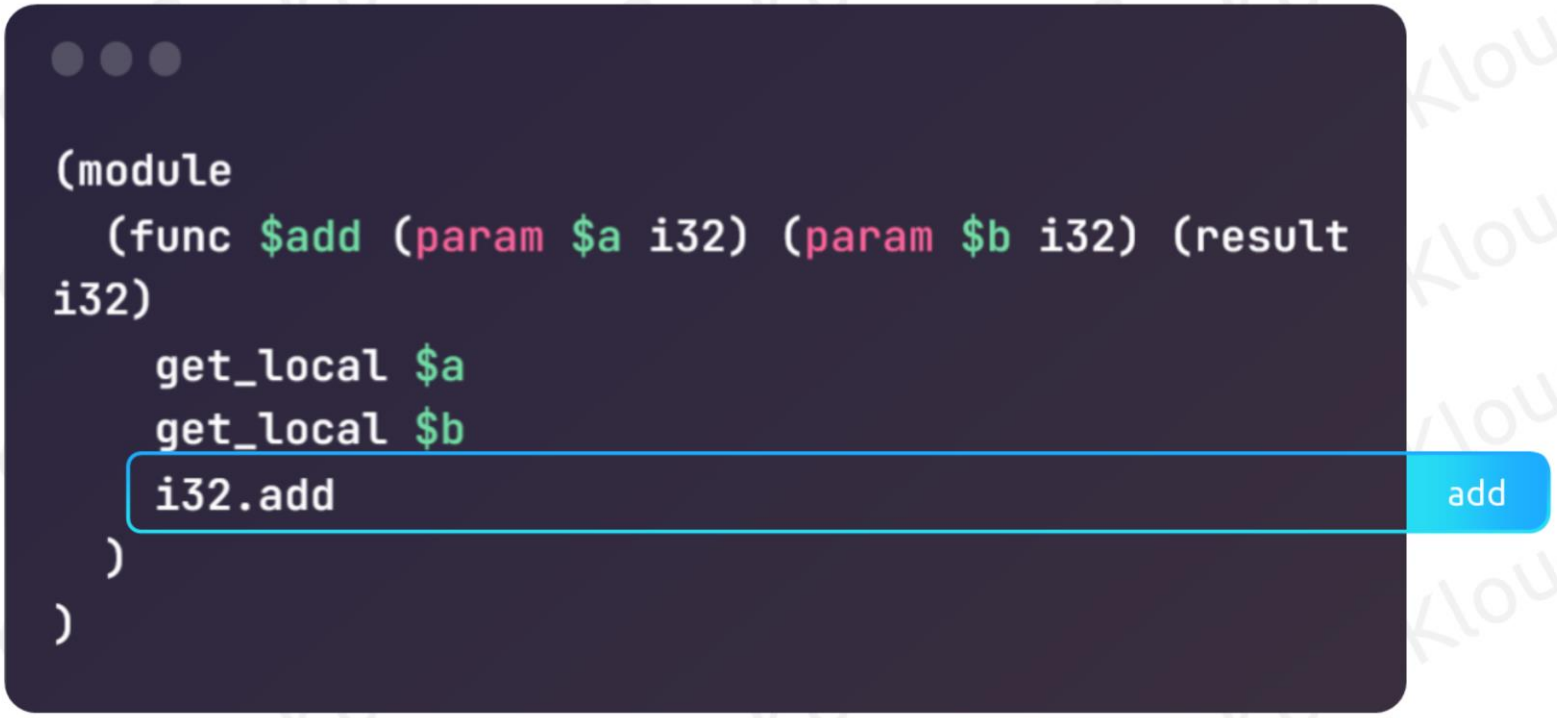
Linear Memory





WebAssembly code is organized into modules. These are like self-contained boxes that have all the functions, variables, and memory declarations an application needs.

Modules

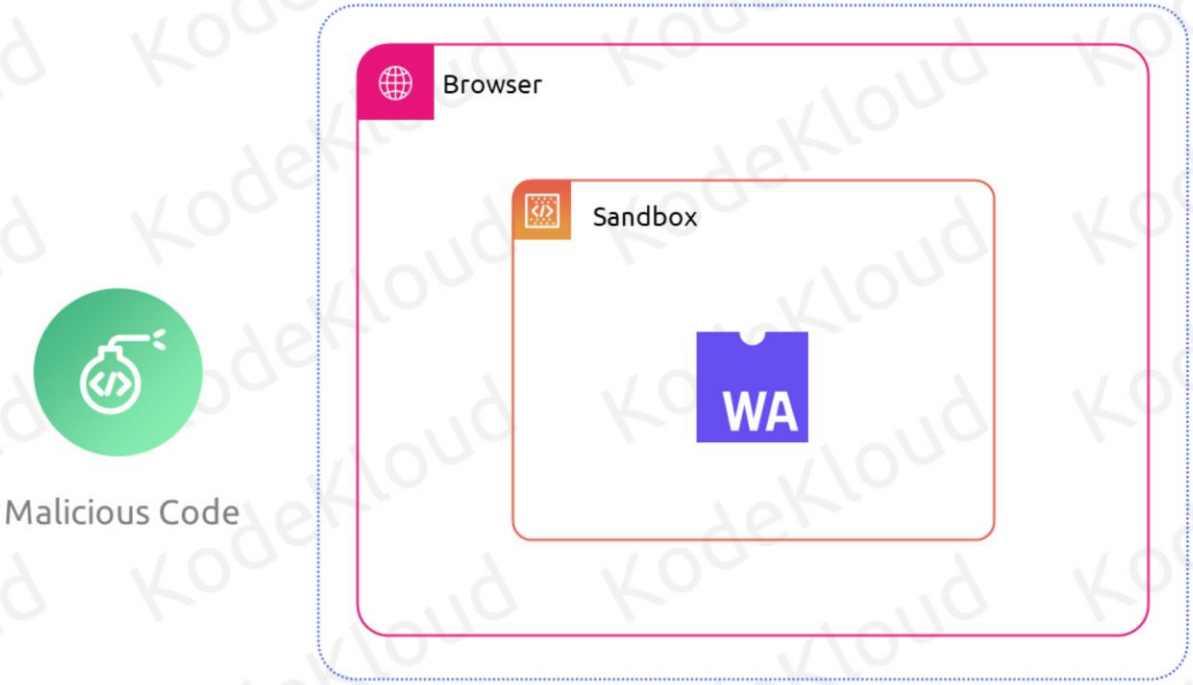


Example:
A WebAssembly module containing our add function might look like:

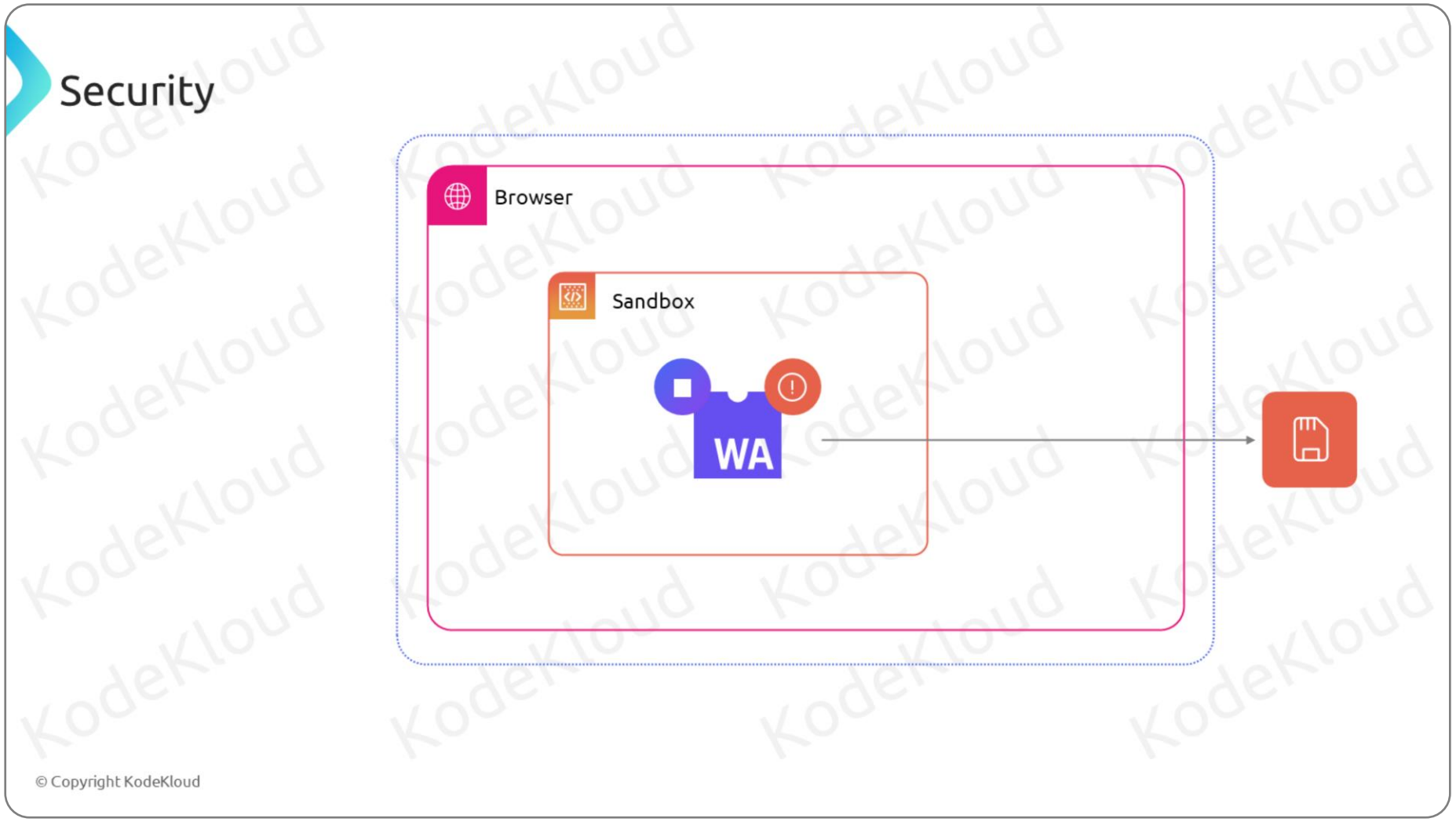
// code //

You can load this module in a browser and call the add function as needed.

Security

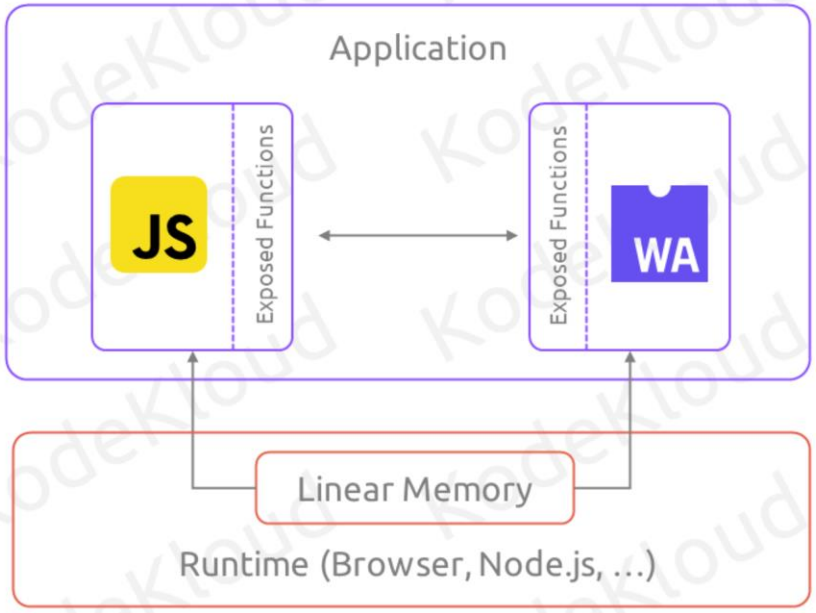


WebAssembly is designed with security in mind. It runs inside a sandbox in the browser, ensuring that even if there's malicious code, it can't harm the user's system.



Example:
If our add function in a WebAssembly program tries to access a memory location it hasn't been given permission to, the browser will halt the program and throw an error. This ensures the safety of the user's data and system.

Interoperability With JavaScript



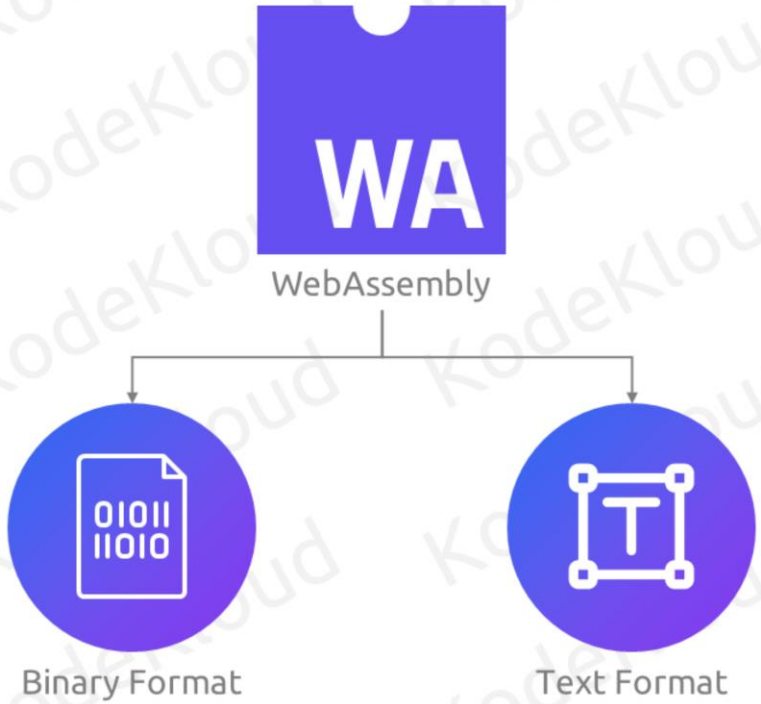

```
Const result = wasmModule.instance.exports.add(3, 4);
console.log(result); // Outputs: 7
```


WebAssembly and JavaScript can work hand-in-hand. You can call WebAssembly functions from JavaScript and vice versa.

Example:
Suppose you have compiled the add function to WebAssembly. In JavaScript, you could call this function like:

// code //

Text Format



While WebAssembly's primary format is binary, there's also a text format. This is a more verbose, human-readable version of the binary, useful for developers to read and debug.

Example:
The binary representation of our add function might be a series of bytes. In the text format, it's more understandable:
// code //

Text Format

While WebAssembly's primary format is binary, there's also a text format. This is a more verbose, human-readable version of the binary, useful for developers to read and debug.

Text Format

```
(func $add (param $a i32) (param $b i32) (result i32)
  get_local $a
  get_local $b
  i32.add
)
```


Example:
The binary representation of our add function might be a series of bytes. In the text format, it's more understandable:
// code //

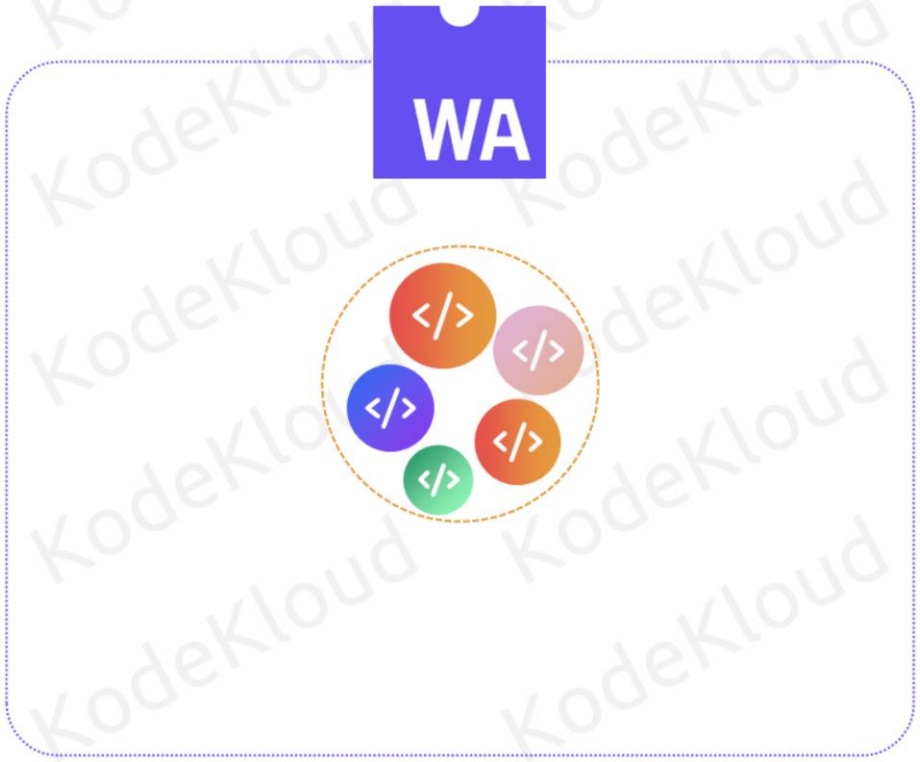
WebAssembly Modules

WebAssembly Modules

If you have learned programming languages in general, functions and modules may not be new to you.

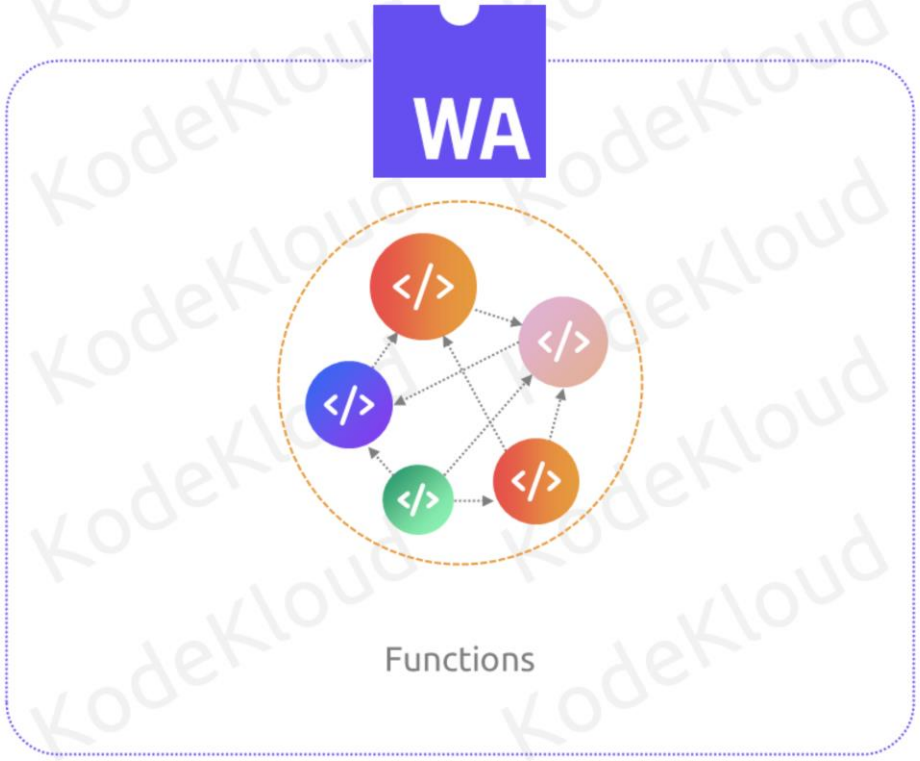
WebAssembly modules stand as the backbone of the WebAssembly ecosystem.

WebAssembly Modules



These modules, precompiled chunks of code, are pivotal in ensuring the structured execution and interaction of WASM code across various platforms.

Functions in WebAssembly



First, let’s understand Functions in WebAssembly.
Functions in WebAssembly are defined blocks of code that can be invoked from other parts of the program or even from JavaScript. They can take parameters and return values, and they reside within the modules, serving as the primary units of execution.

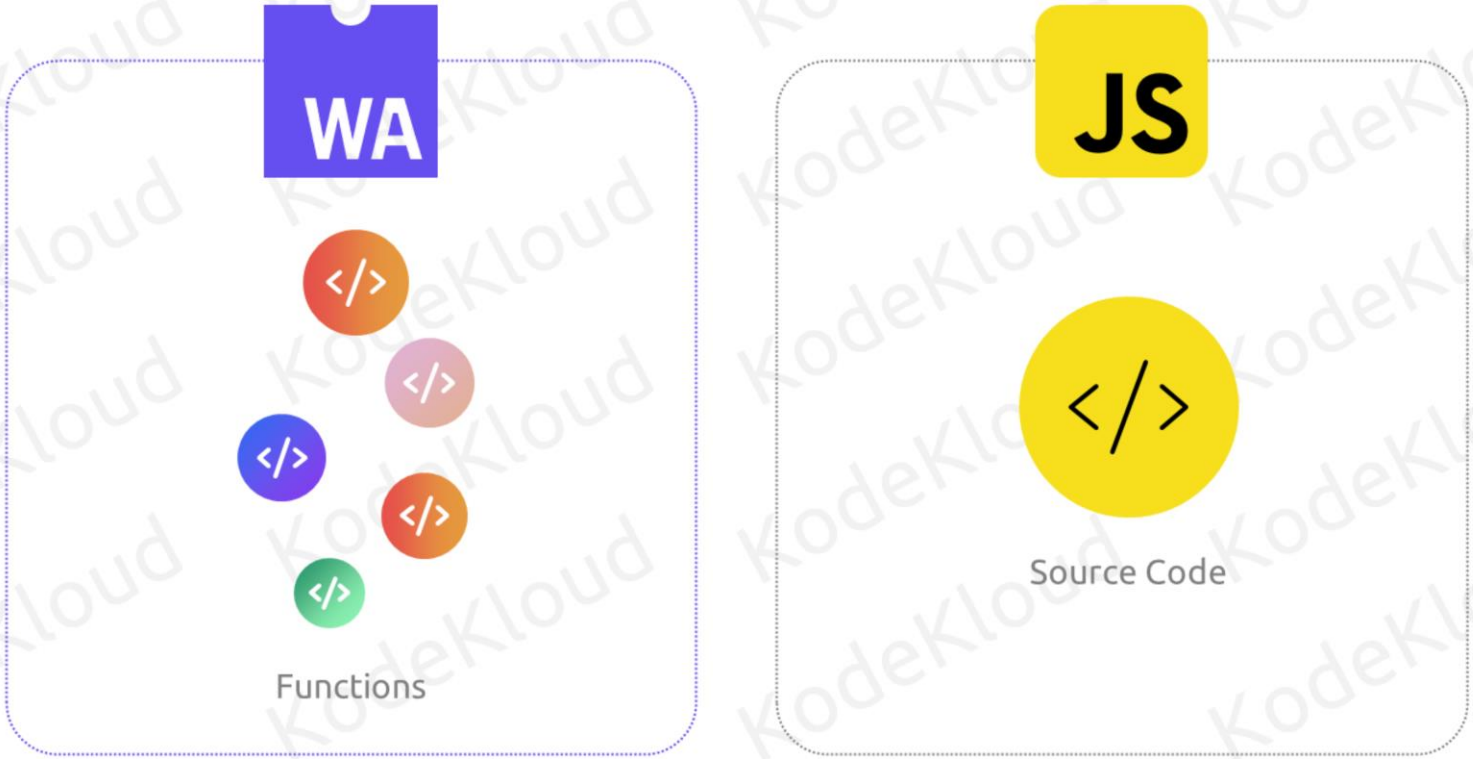
Simple Function Example:

// code //

Consider a function named add that takes two integers as parameters and returns their sum. In WebAssembly's text format, this function might be represented as:

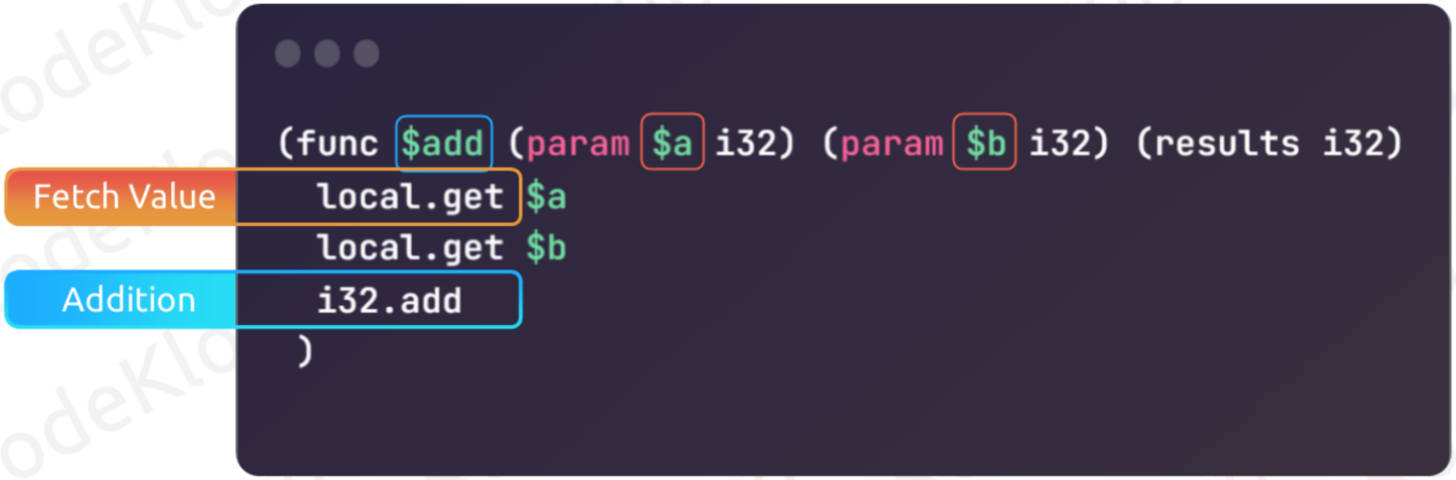
In this example, $add is a function that takes two parameters $a and $b of type i32 (32-bit integer) and returns their sum. The local.get instructions fetch the values of the parameters and i32.add performs the addition.

Functions in WebAssembly



First, let’s understand Functions in WebAssembly.
Functions in WebAssembly are defined blocks of code that can be invoked from other parts of the program or even from JavaScript. They can take parameters and return values, and they reside within the modules, serving as the primary units of execution.

Functions in WebAssembly



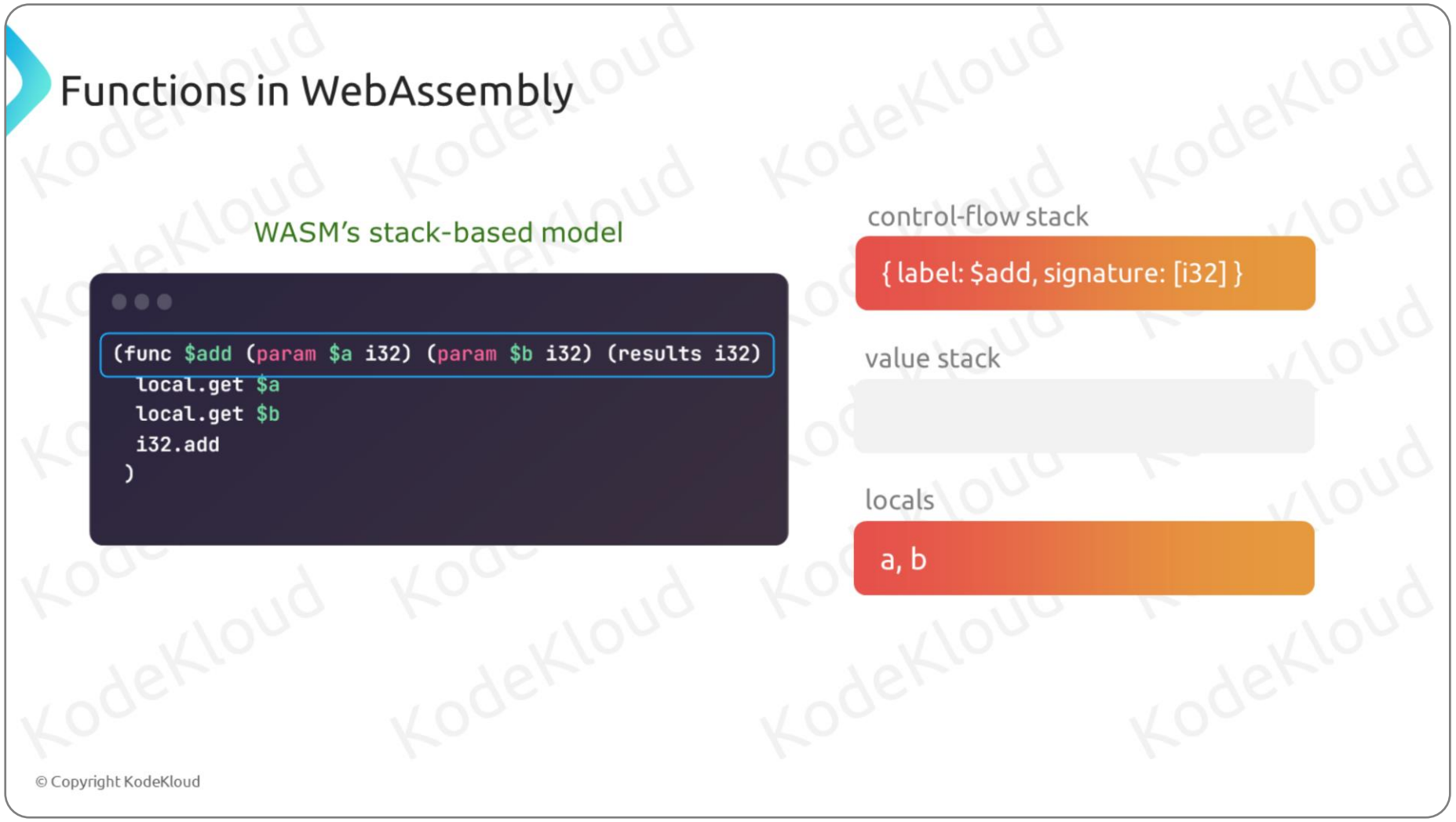
Simple Function Example:

// code //

Consider a function named add that takes two integers as parameters and returns their sum. In WebAssembly's text format, this function might be represented as:

In this example, $add is a function that takes two parameters $a and $b of type i32 (32-bit integer) and returns their sum. The local.get instructions fetch the values of the parameters and i32.add performs the addition.

Functions in WebAssembly



In WASM's stack-based model, The result of the last expression in a function's body is implicitly returned if the function is defined with a result type. There's no return keyword as in many high-level languages.

// code //

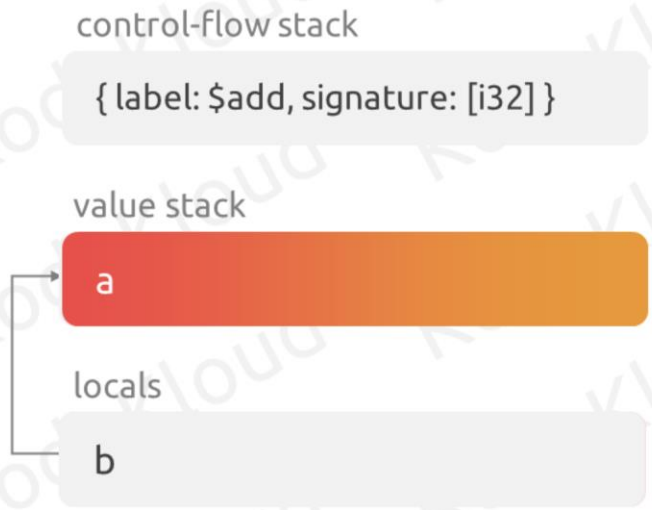
Here, local.get $a and local.get $b push the values of $a and $b onto the stack, and i32.add pops these two values, adds

them, and pushes the result back onto the stack. Since i32.add is the last instruction and the function declares a result type of i32, the value on top of the stack is returned.

Functions in WebAssembly

WASM's stack-based model

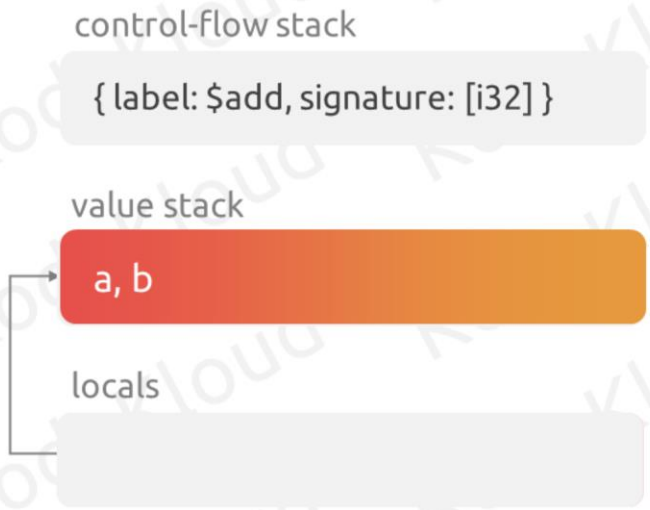
```
(func $add (param $a i32) (param $b i32) (results i32)
 local.get $a
 local.get $b
 i32.add
)
```



Functions in WebAssembly

WASM's stack-based model

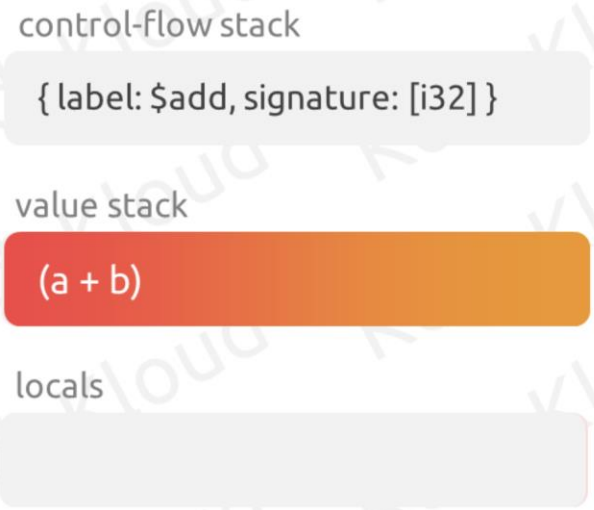
```
(func $add (param $a i32) (param $b i32) (results i32)
  local.get $a
  local.get $b
  i32.add
 )
```



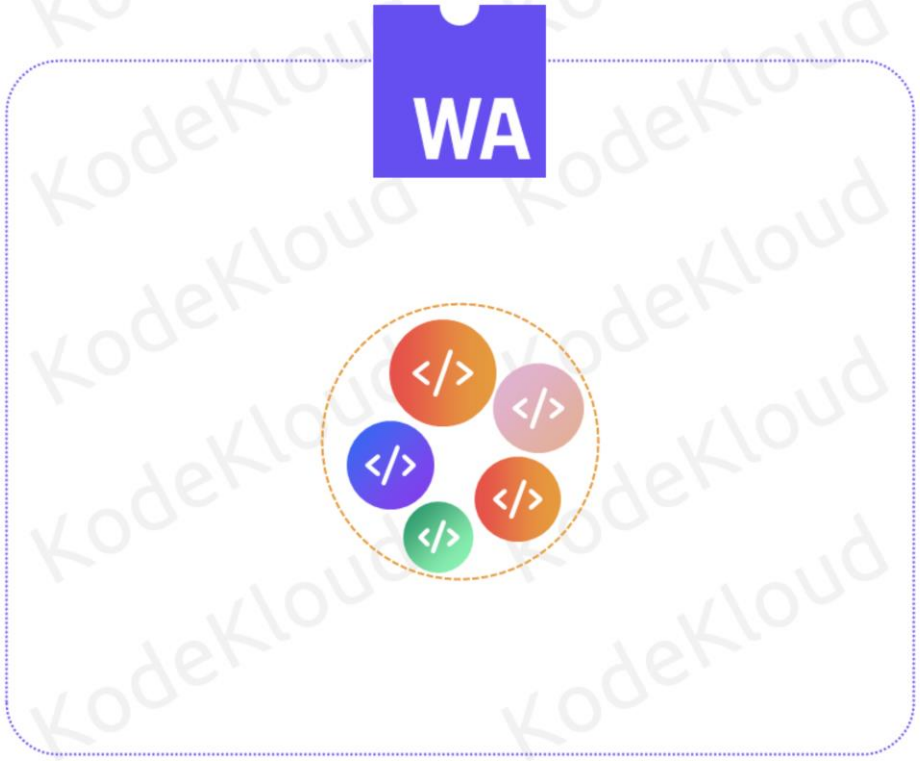
Functions in WebAssembly

WASM's stack-based model

```
(func $add (param $a i32) (param $b i32) (results i32)
  local.get $a
  local.get $b
  i32.add
 )
```

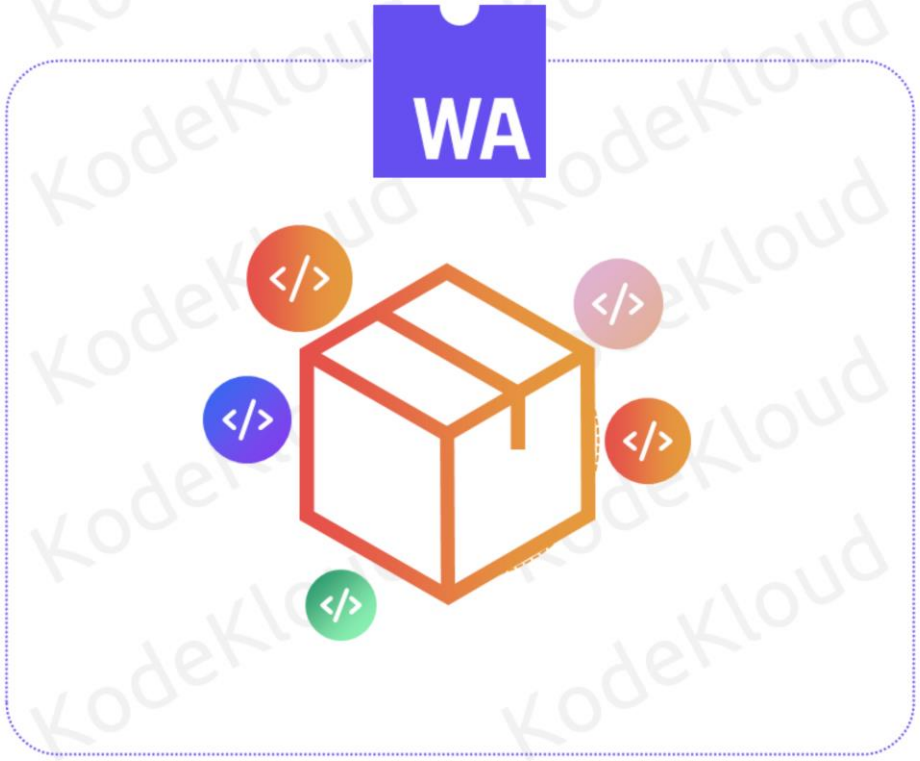


The Essence of WebAssembly Modules



Building upon the concept of functions, a WebAssembly module serves as a cohesive unit that houses these functions and provides a structured environment for their execution.

The Essence of WebAssembly Modules



It's not just about individual functions; it's about how these functions, along with other components, come together to form a complete, executable module.

At its core, a WebAssembly module is a container for binary code and associated data. It's akin to a sealed box that holds all the necessary components for a specific functionality.

The Essence of WebAssembly Modules

```
(module
  (func $add (param $a i32) (param $b i32) (result I32)
    get_local $a
    get_local $b
    i32.add)
   (func $subtract ...)
   (func $multiply ...)
)
```


//code//

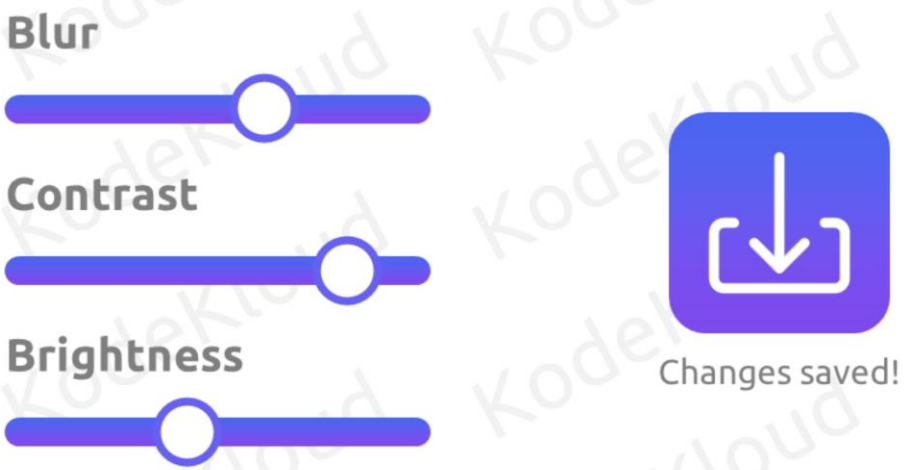
Example: Imagine a module designed for mathematical operations. This module would house functions for addition, subtraction, multiplication, and so forth.
I hope we have explained about "functions" before this module. If so, then we can easily build on top of that here. Something like group multiple functions together and you get a module.

it should be explained here. I've done that before the modules section

Beyond Functions

Apart from functions, WebAssembly modules also encapsulate several other key components: they include global variables, which are accessible throughout the module; tables, which are essentially arrays holding references to functions, allowing for dynamic function invocation; and memories, which are resizable linear arrays of bytes for data storage. Soon, we will delve deeper into understanding how tables and memory work.

Module for Image Processing

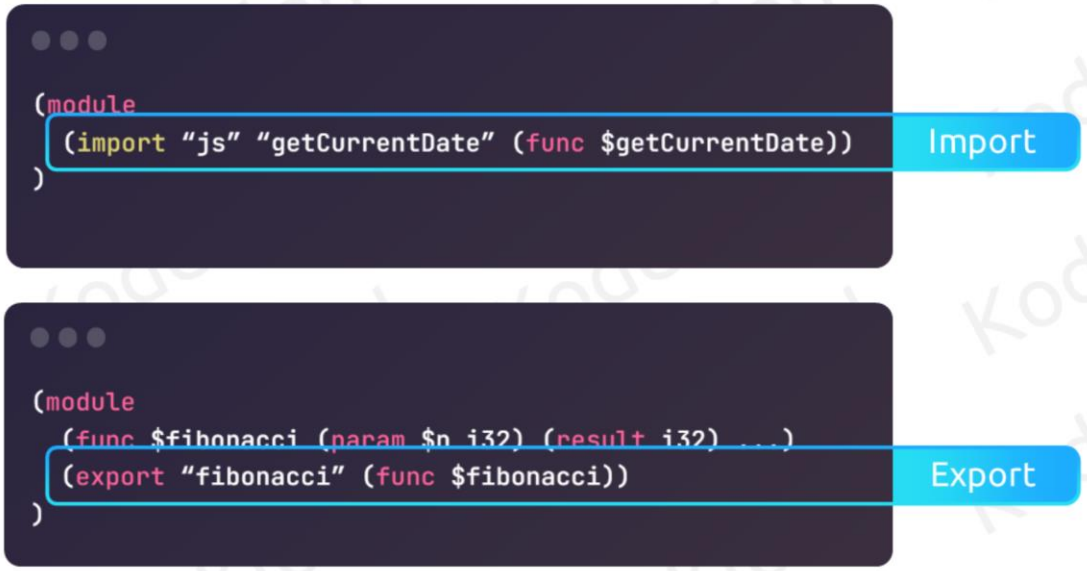


In an image processing WebAssembly module, you would typically find functions designated for various image filters (like blur or contrast adjustment).

Module for Image Processing

Additionally, the module would include a table that maps filter names to their corresponding functions, facilitating dynamic selection and application of these filters. Lastly, there's a memory segment within the module, which is used for storing the raw image data that these functions operate on.

Interactivity – Imports and Exports



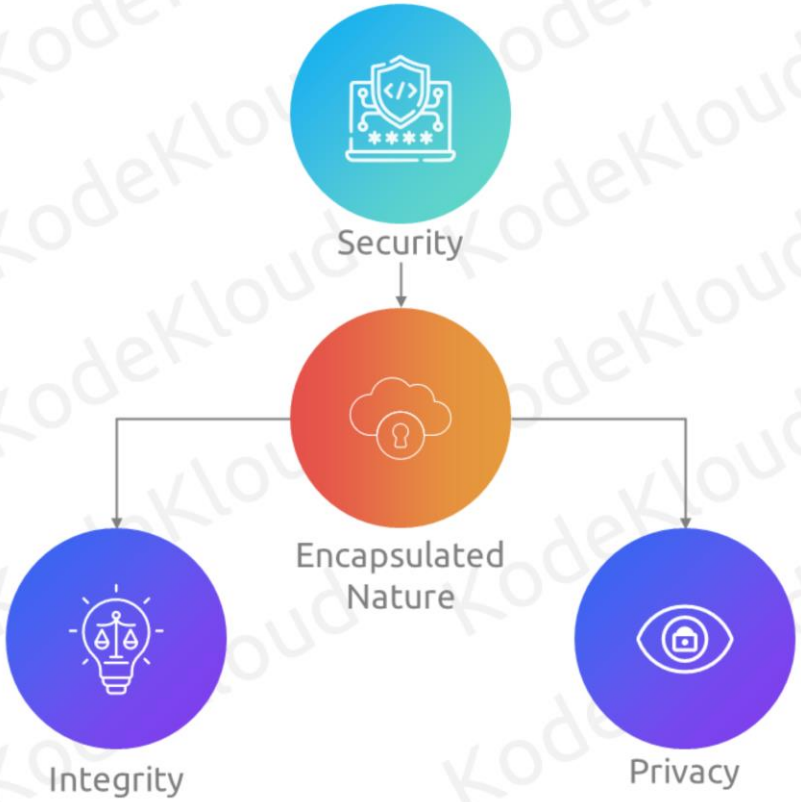
//code//
Imports
To be versatile, modules can declare dependencies, pulling in necessary functions or variables from their environment.

If a module requires the current date from JavaScript, it would import a function like so:

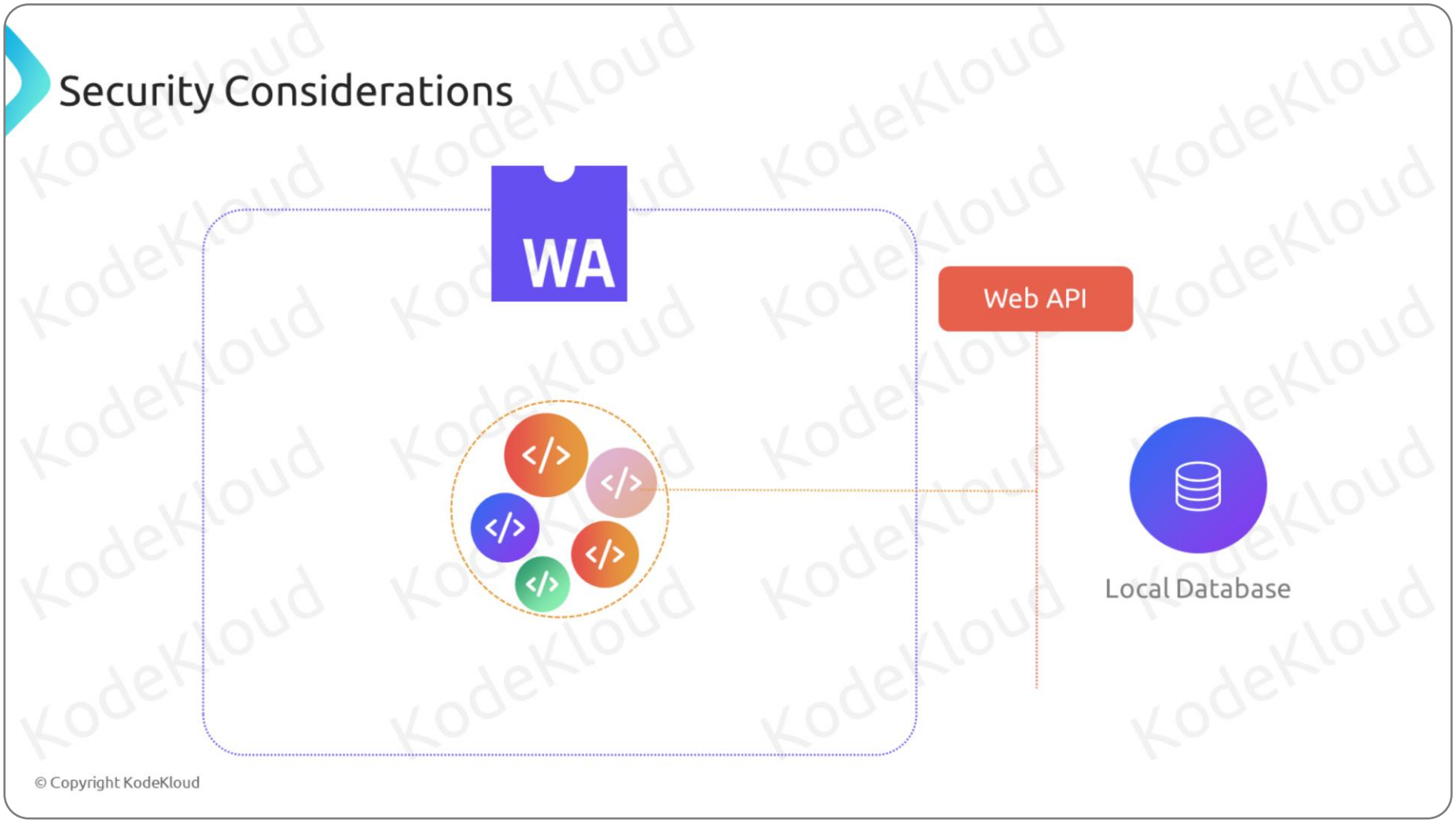
//code//
Exports
Conversely, to offer its functionalities to the outside world, a module can export specific components.

To make a Fibonacci calculator accessible:

Security Considerations



WebAssembly modules are designed with a strong emphasis on security. Their encapsulated nature ensures that they operate within a confined environment, preventing unintended side effects.



Even if a module has a function intended to access a database, it can't directly reach out to a user's local database without interfacing through specific Web APIs, ensuring data integrity and privacy.

Portability and Universality

WebAssembly's design philosophy emphasizes cross-platform compatibility. Modules, being platform-agnostic, can run anywhere WebAssembly is supported.

A physics simulation module, once created, can be executed on a PC's web browser, a smartphone's app, or even a smart TV, ensuring consistent behavior and performance.

WebAssembly Ecosystem – Modular Architecture

The modular nature of WebAssembly promotes code reusability and maintainability. By segmenting applications into distinct modules, developers can achieve efficient and organized codebases.

Complex Web Application

3D Model Editor

Geometry Calculations

Texture Mapping

User Interface

For a complex web application like an online 3D model editor, different modules could handle geometry calculations, texture mapping, and user interface interactions..

Complex Web Application

3D Model Editor

Texture Mapping

Geometry Calculations

User Interface

When a new shading technique emerges, only the relevant module needs updating, leaving others untouched.

Instructions and Data Types

Introduction

In the world of programming, understanding instructions and data types is akin to knowing the alphabet and grammar of a language. Before we dive into the specifics of WebAssembly's instructions and data types, let's first address the fundamental question: Why are they crucial?

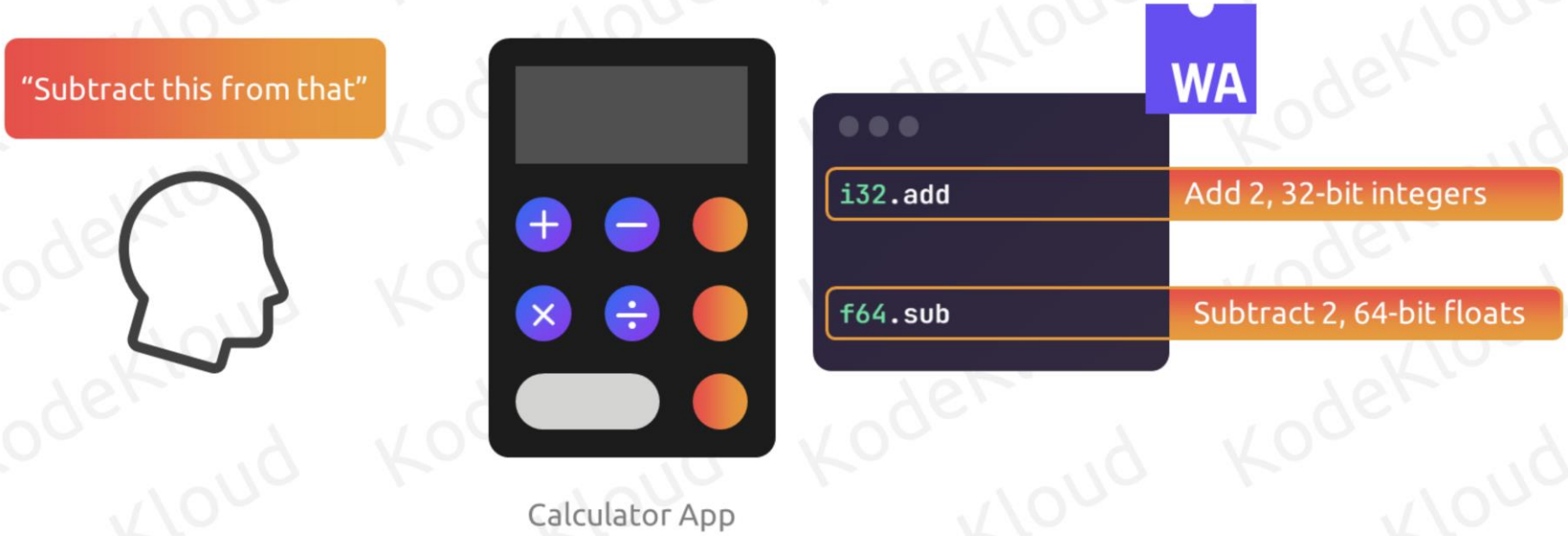
Instructions and Data Types

At its core, WebAssembly is about executing code efficiently in a web environment. To achieve this, it uses a set of specific instructions and data types. Think of instructions as the verbs of a language, dictating what actions to perform, and data types as the nouns, representing the kinds of data we work with. Together, they form the foundation of any WebAssembly program, ensuring it runs smoothly and predictably.

WebAssembly Instructions

What are they?
Instructions are the basic commands that WebAssembly understands. They dictate the operations to be performed.

WebAssembly Instructions



Example: Imagine you're building a calculator app. In human terms, you'd want operations like "add these numbers" or "subtract this from that." In WebAssembly, these operations translate to instructions like i32.add for adding two 32-bit integers or f64.sub for subtracting two 64-bit floats.

WebAssembly Instructions

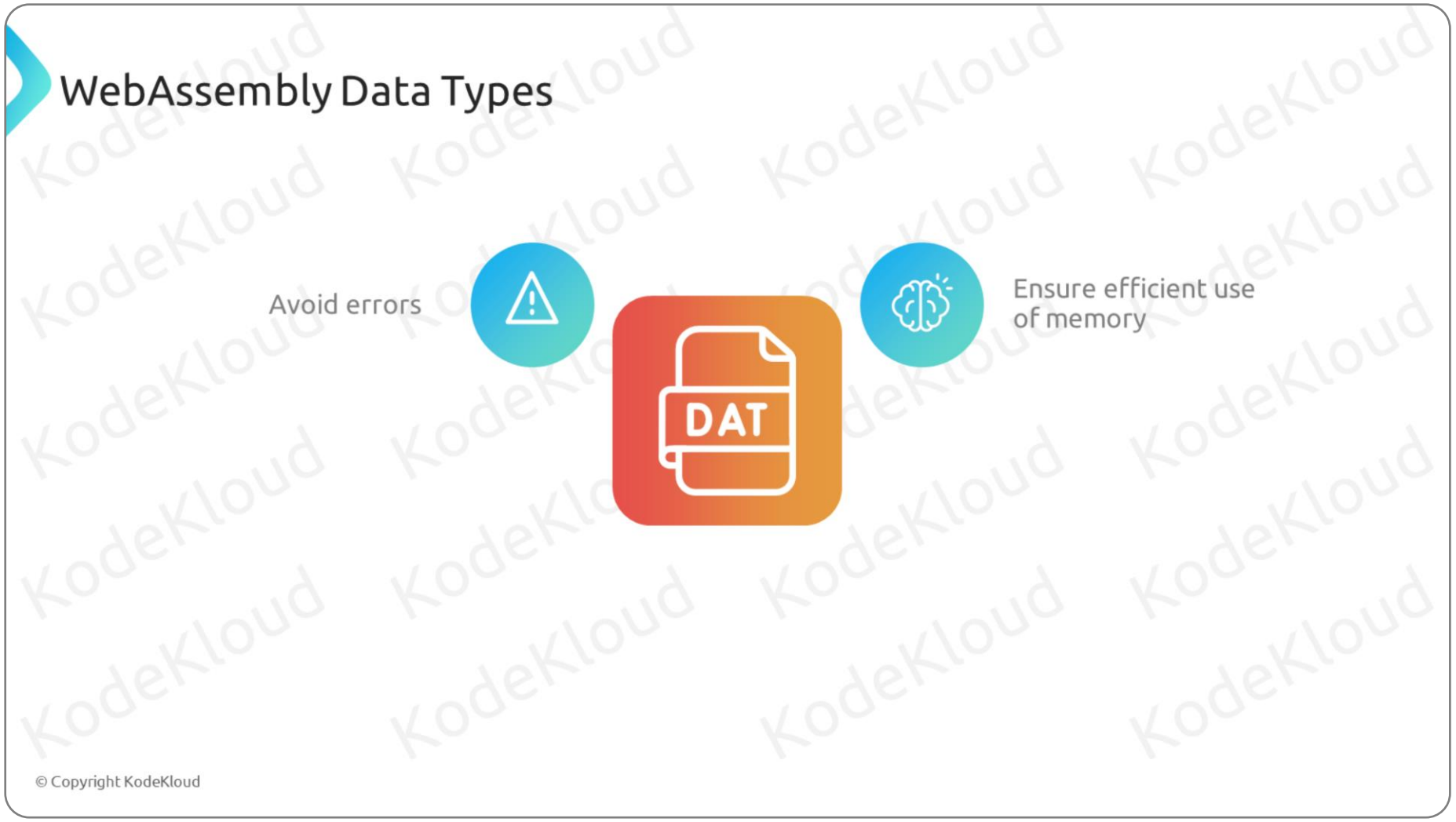
Why are they unique?
WebAssembly instructions are low-level, meaning they're closer to machine code than high-level programming languages. This ensures faster execution and better optimization by browsers.

WebAssembly Data Types

What are they?
Data types define the kind of data we can work with in WebAssembly. They set the rules for how data is stored and processed.

WebAssembly Data Types

Example: Consider a weather app displaying temperatures. While the temperature in one city might be a whole number (like 25°C), another city might have a decimal value (like 25.5°C). In WebAssembly, the former could be represented using the i32 data type (integer) and the latter with f32 (floating-point number).



Why are they essential?
Data types ensure that our WebAssembly code is predictable. By defining the kind of data and how it's used, we avoid errors and ensure efficient use of memory.

Imagine pouring water into containers of different shapes. Each container (data type) can hold a specific amount and shape of water (data). If we try to pour too much water or the wrong type of liquid, it either won't fit or won't function as

intended. Similarly, using the right data type ensures our data fits well and works correctly.

WebAssembly Data Types

1 Liter

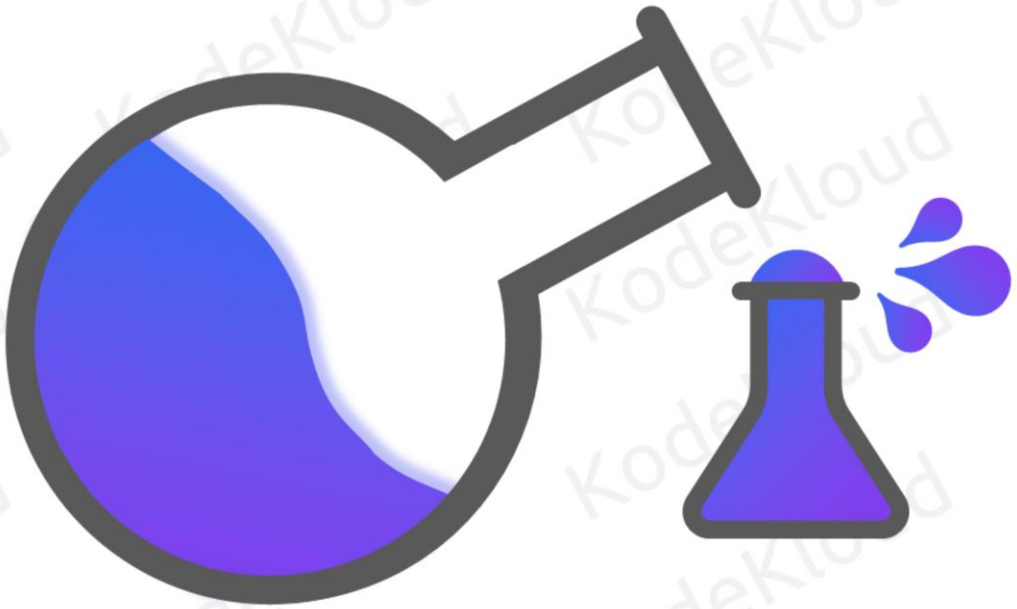
Food Coloring (Baking)

300 mL

Cooking Oil (Frying)

Imagine pouring water into containers of different shapes. Each container (data type) can hold a specific amount and shape of water (data).

WebAssembly Data Types



If we try to pour too much water or the wrong type of liquid, it either won't fit or won't function as intended. Similarly, using the right data type ensures our data fits well and works correctly.

A Symbiosis of Instructions and Data Types

Instructions and data types in WebAssembly are intertwined. Each instruction expects data of specific types. This relationship ensures that operations are performed correctly and efficiently.

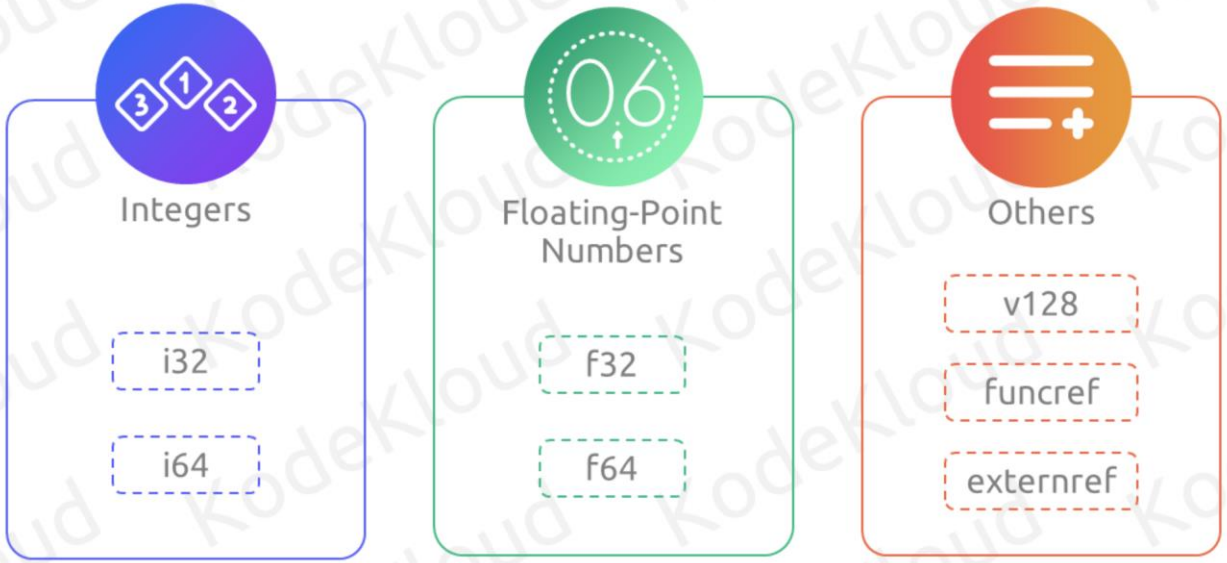
A Symbiosis of Instructions and Data Types

Example: Let's go back to our calculator app. If a user wants to multiply two numbers, the instruction might be f64.mul. This instruction specifically expects two 64-bit floating-point numbers. If we mistakenly provide it with text or a boolean value, it wouldn't work. Because the operation doesn't make sense.

This following reference list provides a snapshot of some of the fundamental data types and instructions in WebAssembly. It's by no means exhaustive, but it should serve as a handy reference for students as they delve deeper into WebAssembly's

intricacies.

WebAssembly Data Types – References

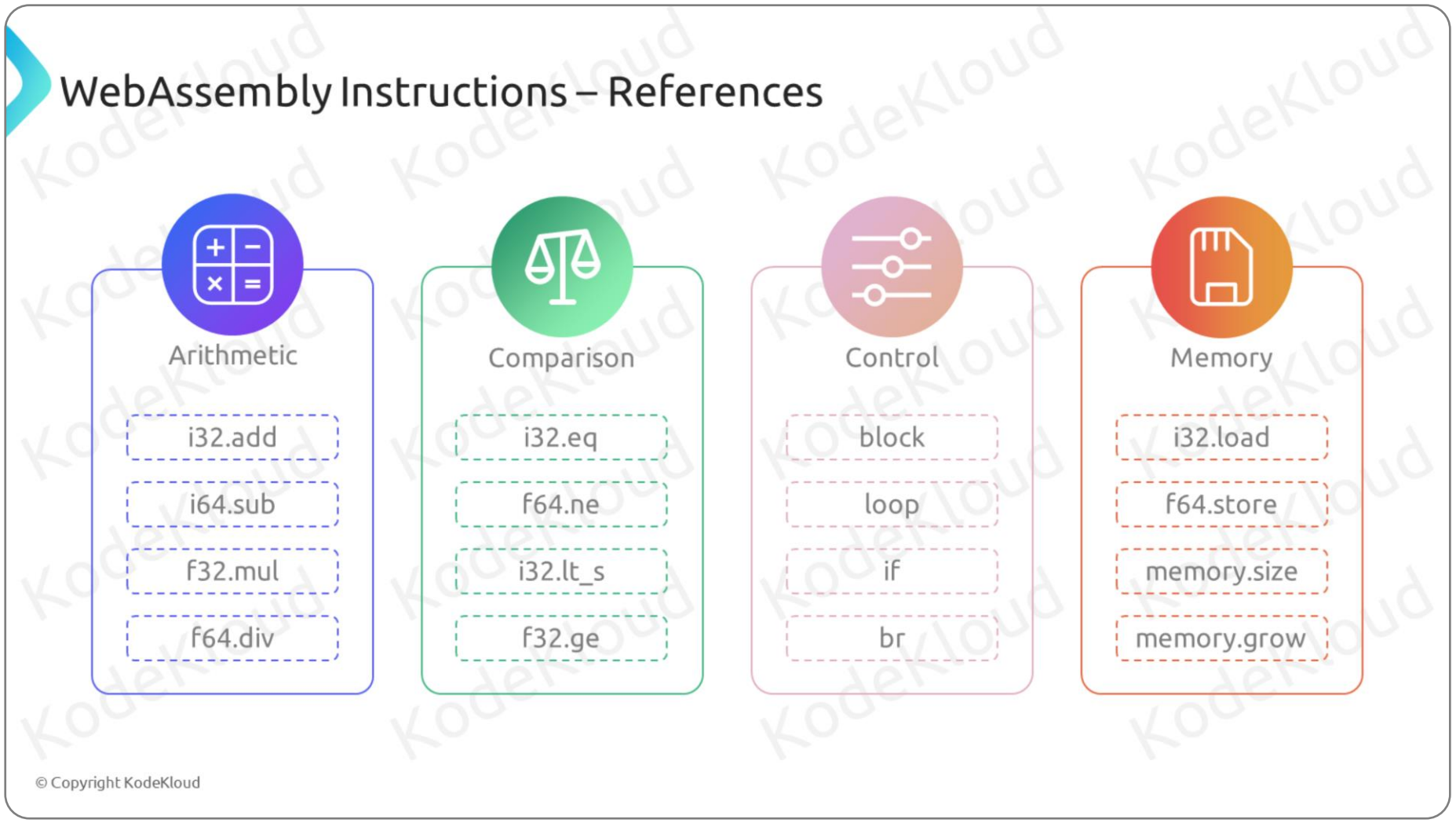


Integers:

i32: 32-bit integer. Useful for most numerical operations.
i64: 64-bit integer. Used for larger numerical values or high-precision calculations.

Floating-Point Numbers:

f32: 32-bit floating-point number. Suitable for decimal values with moderate precision.
f64: 64-bit floating-point number. Ideal for decimal values requiring high precision.
Others:
v128: SIMD (Single Instruction, Multiple Data) vector type, used for parallel processing.
funcref: A reference to a function.
externref: A reference to an external value, often used for interoperability with JavaScript.



Arithmetic Instructions:
i32.add: Add two 32-bit integers.
i64.sub: Subtract one 64-bit integer from another.
f32.mul: Multiply two 32-bit floats.
f64.div: Divide one 64-bit float by another.
Comparison Instructions:

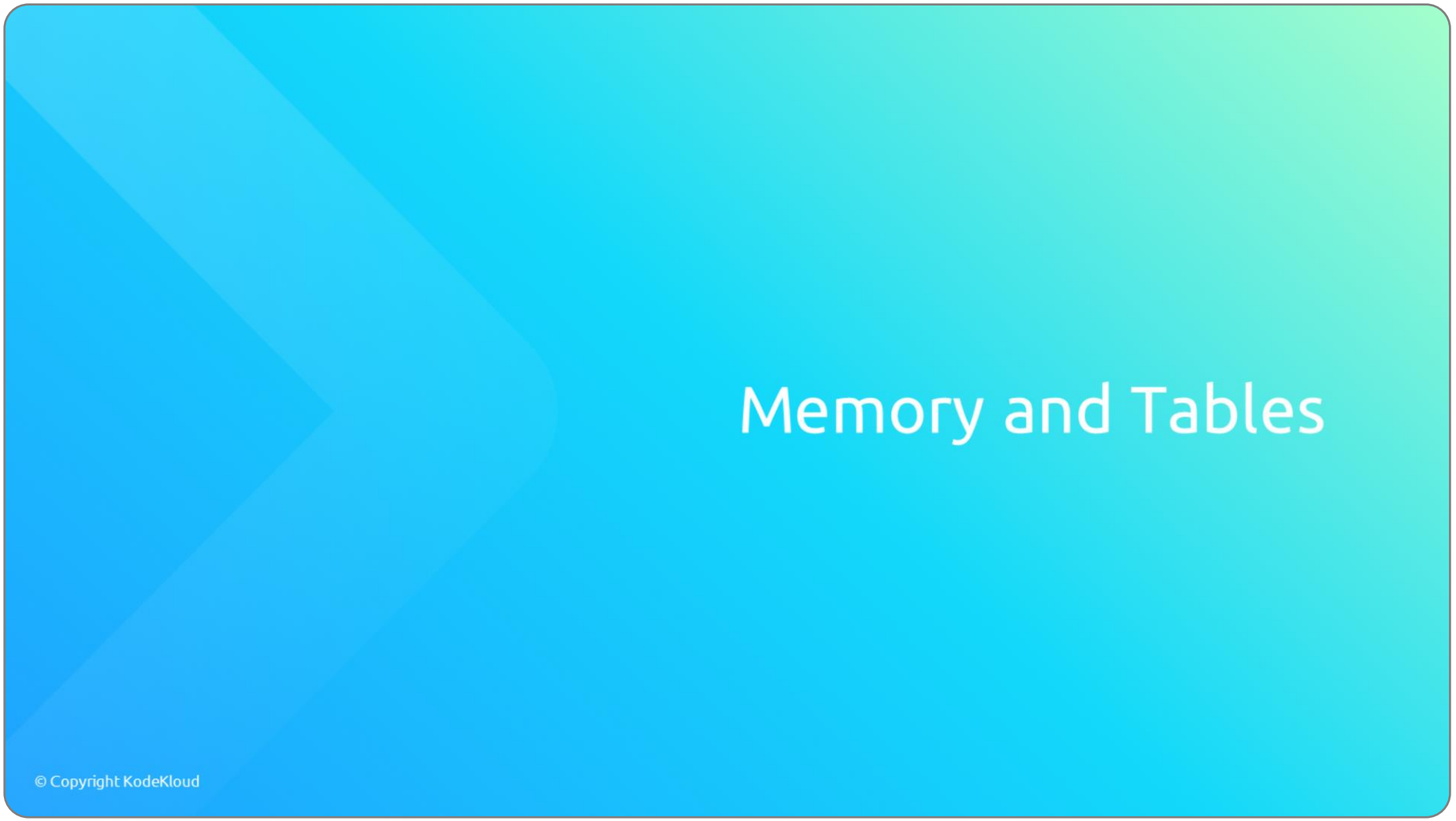
i32.eq: Check if two 32-bit integers are equal.
f64.ne: Check if two 64-bit floats are not equal.
i32.lt_s: Check if one signed 32-bit integer is less than another.
f32.ge: Check if one 32-bit float is greater than or equal to another.
Control Instructions:
block: Begin a sequence of instructions.
loop: Create a loop.
if: Conditional execution based on a test.
br: Branch to a given label.
Memory Instructions:
i32.load: Load a 32-bit integer from memory.
f64.store: Store a 64-bit float to memory.
memory.size: Get the current size of memory.
memory.grow: Increase the size of memory.

WebAssembly Instructions – References

Official WebAssembly Specification: WebAssembly Core Specification

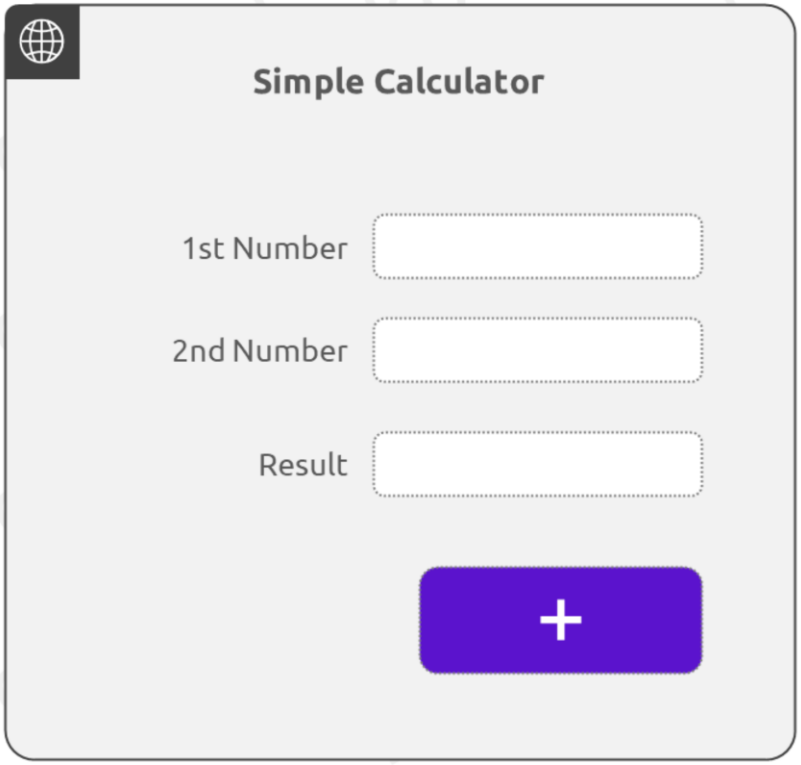
MDN Web Docs on WebAssembly: MDN WebAssembly Reference

For the most up-to-date and comprehensive information, please refer to the official WebAssembly documentation and the MDN Web Docs. These resources provide a wealth of detailed explanations and examples, ensuring you have access to the latest data types,instructions, and conventions in WebAssembly development.



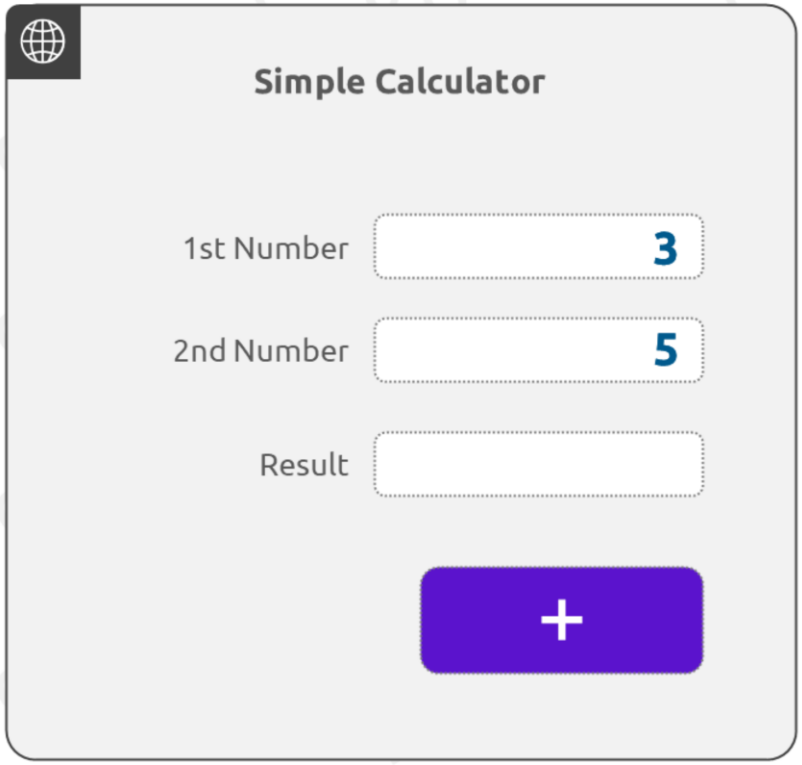
Imagine we have a simple computer program, or a WebAssembly module, that's designed to do math operations like addition, subtraction, multiplication, and division. This program is like a small calculator that can run in a web browser.

Step 1: Storing Data in WebAssembly Memory



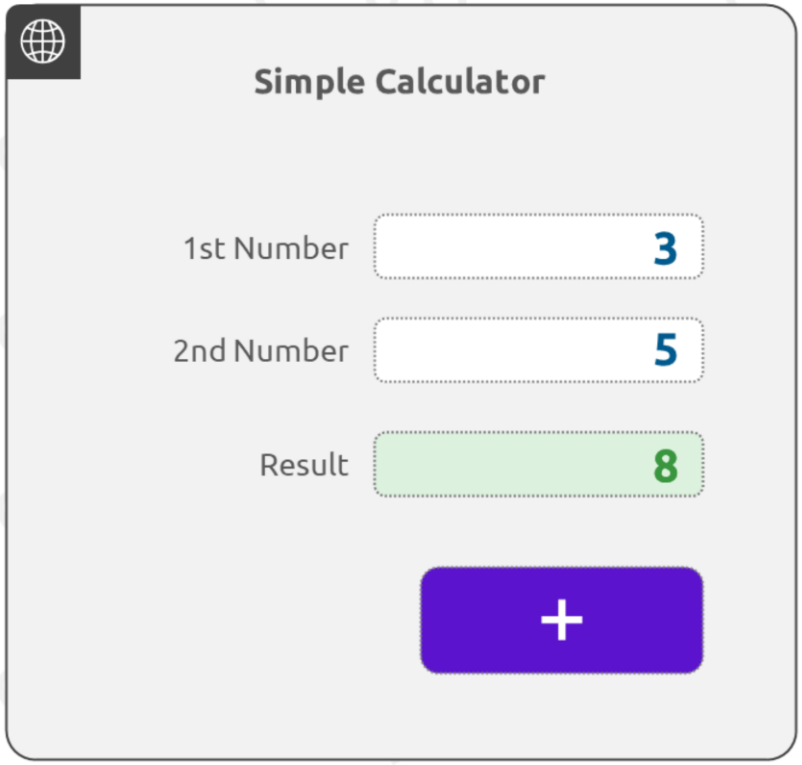

```c
#include <emscripten.h>

extern "C" {
  int add(int a, int b) {
    return a + b;
  }

  int calculate(int op, int a, int b) {
    switch (op) {
      case 0:
        return add(a, b);
      default:
        return 0;
    }
  }
}
```

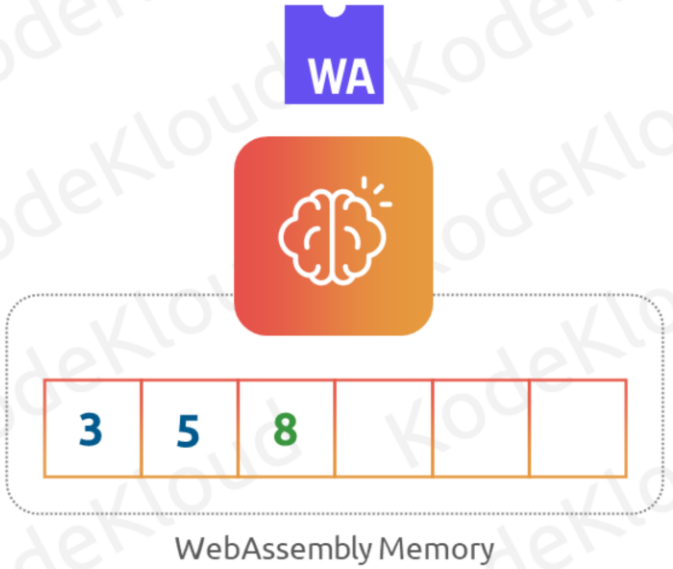
Step 1: Storing Data in WebAssembly Memory



When you use this calculator to add two numbers, say 3 and 5, these numbers are stored in what's called WebAssembly memory. This memory is like a series of boxes where each box can hold a piece of data.

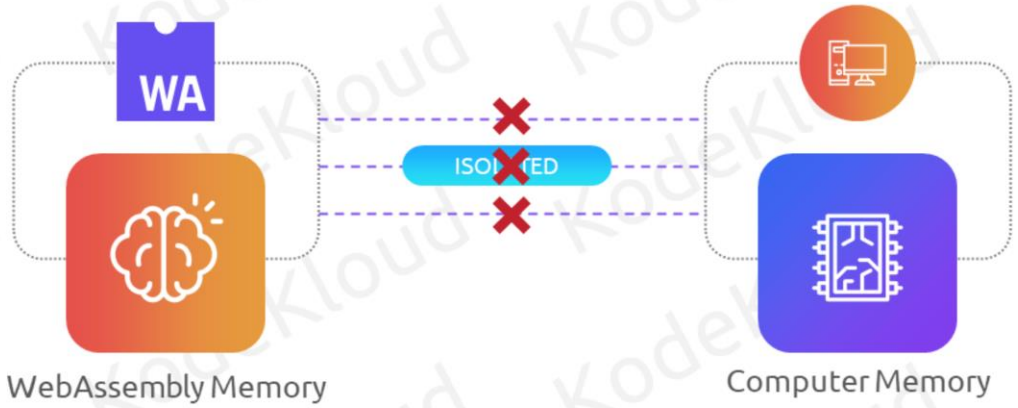
Step 1: Storing Data in WebAssembly Memory





After the calculation (3 + 5), the result, which is 8, is also stored in this memory. So, WebAssembly memory is where all our numbers and results are kept.

Step 1: Storing Data in WebAssembly Memory

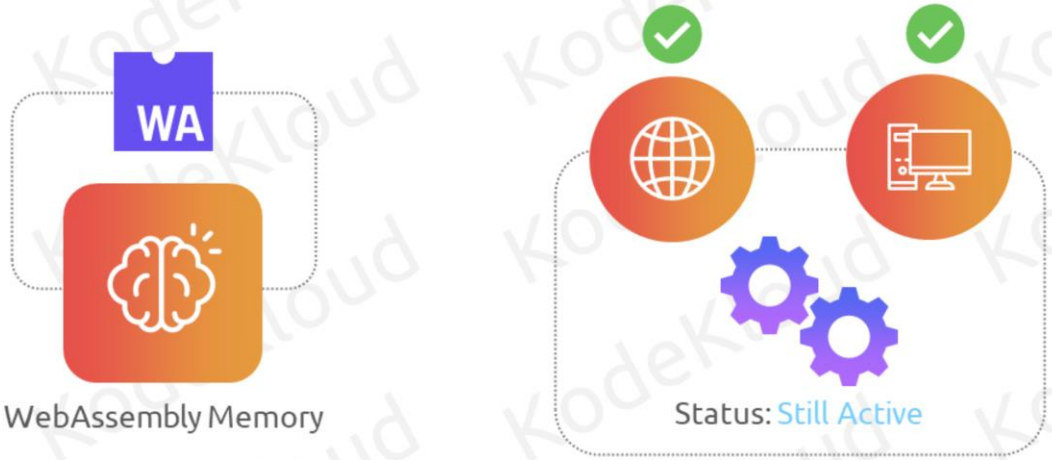


This memory is isolated from the rest of the computer's memory, which makes it secure. If something goes wrong in our calculator, it won't affect the rest of the computer or the web browser.

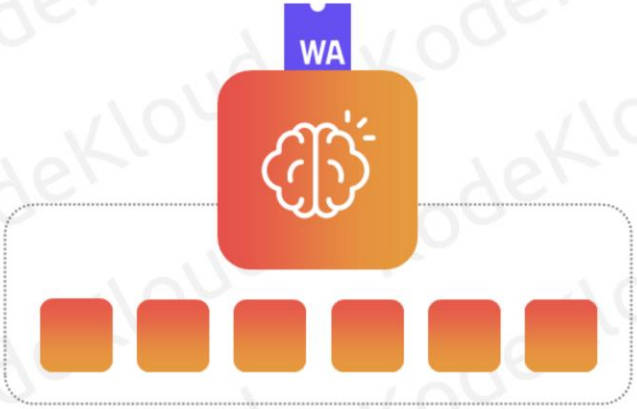
Step 1: Storing Data in WebAssembly Memory

Also, the memory can grow as needed. If our calculations need more space, the program can increase the number of boxes available.

Step 1: Storing Data in WebAssembly Memory

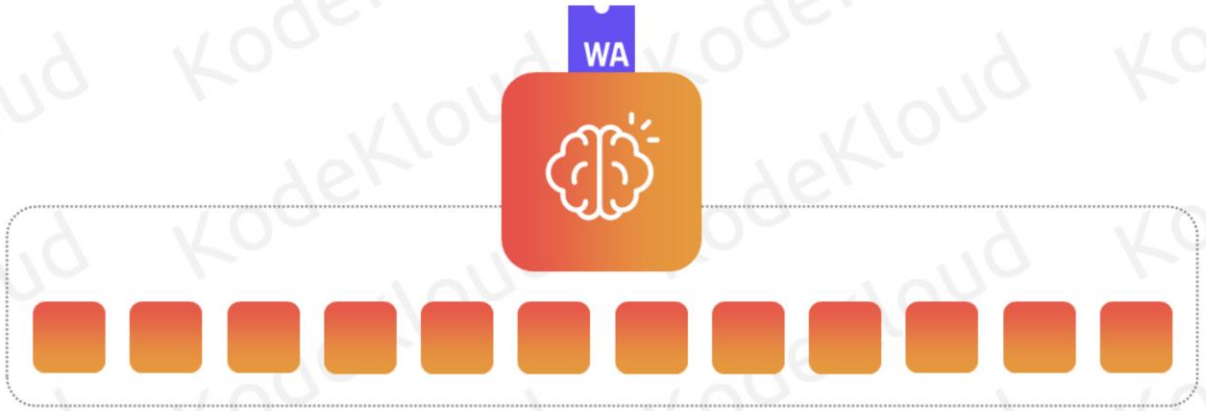


Step 1: Storing Data in WebAssembly Memory



Memory can grow as needed.

Step 1: Storing Data in WebAssembly Memory



Memory can grow as needed.

Step 2: Using WebAssembly Tables for Functions

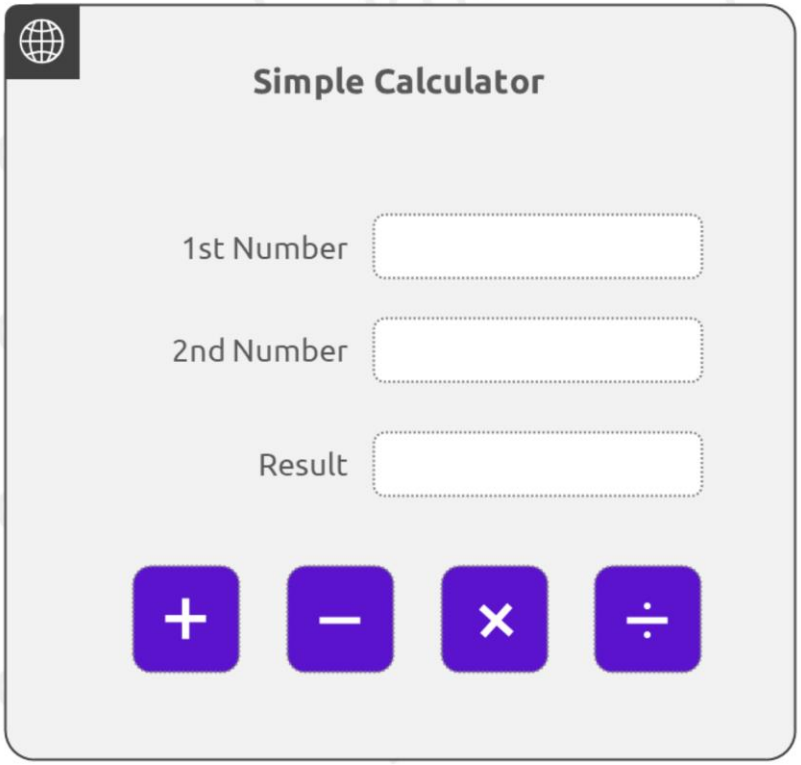
WASM Table

Step 2: Using WebAssembly Tables for Functions


```c
#include <emscripten.h>

extern "C" {
  int add(int a, int b) {
    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }

  int multiply(int a, int b) {
    return a * b;
  }

  int divide(int a, int b) {
    return a / b;
  }

  int calculate(int op, int a, int b) {
    typedef int (*op_func)(int, int);
    op_func table[] = {add, subtract, multiply, divide};
    op_func func = table[op];
    int result = func(a, b);
    return result;
  }
}
```


Now, our calculator needs to know how to perform each math operation. This is where WebAssembly tables come into play. They are like a directory that lists where each operation (like add, subtract) is located in the program.

Step 2: Using WebAssembly Tables for Functions

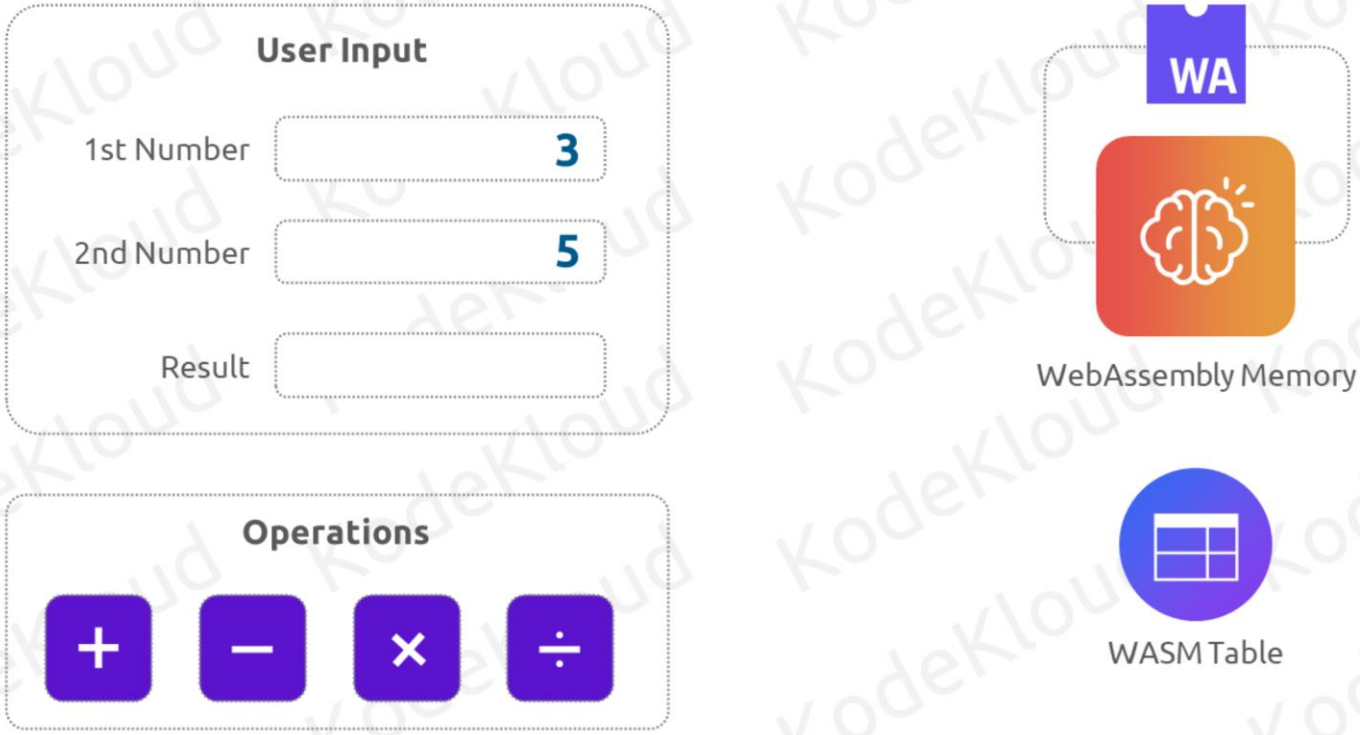



```c
#include <emscripten.h>

extern "C" {
  int add(int a, int b) {
    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }

  int multiply(int a, int b) {
    return a * b;
  }

  int divide(int a, int b) {
    return a / b;
  }

  int calculate(int op, int a, int b) {
    typedef int (*op_func)(int, int);
    op_func table[] = {add, subtract, multiply, divide};
    op_func func = table[op];
    int result = func(a, b);
    return result;
  }
}
```


When we want to do an addition, the program looks in the table, finds where the addition function is, and uses it to perform the calculation.

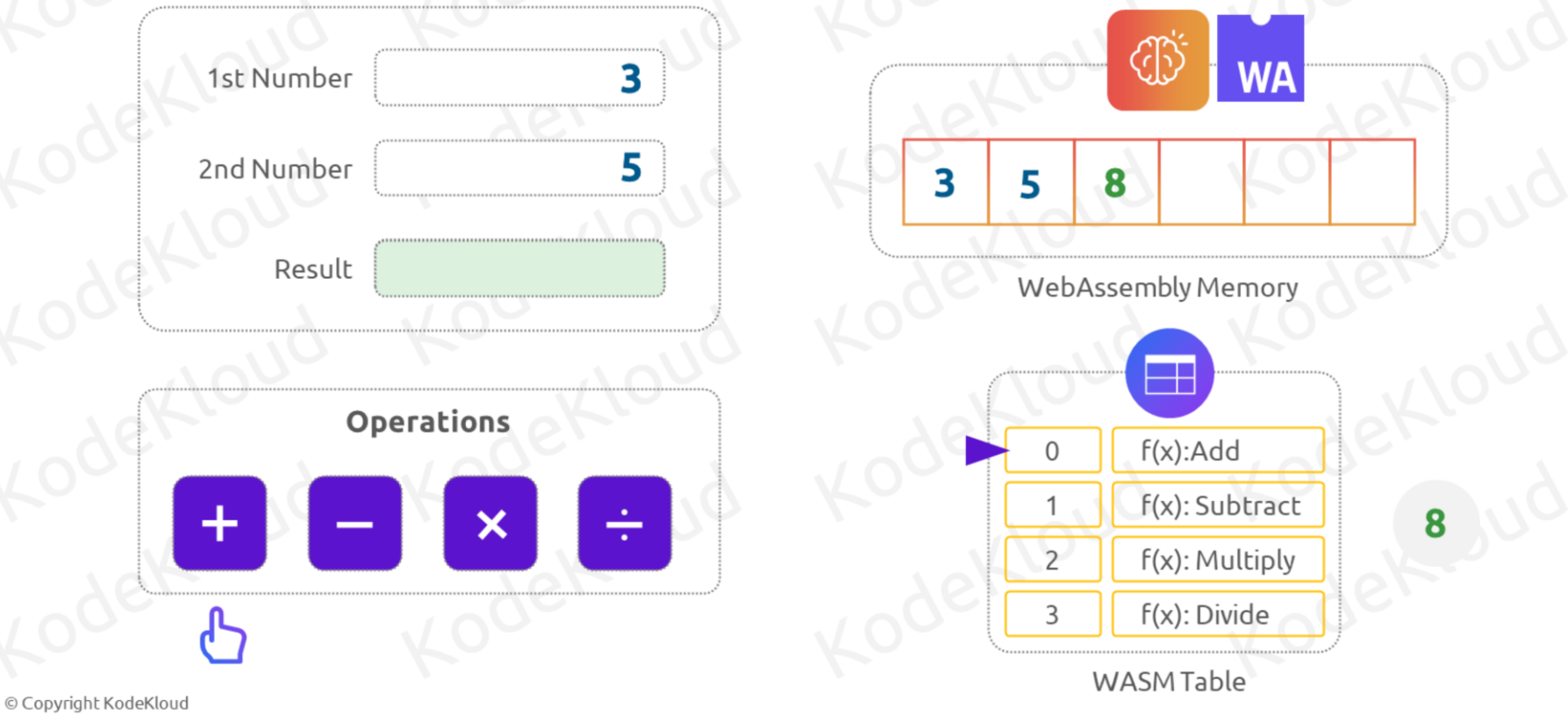
This system allows our calculator to be dynamic. It can handle different operations easily because the table tells it where to find the code for each operation.

Step 2: Using WebAssembly Tables for Functions



Let's say a user on a website inputs 3 and 5 and selects to add them. Our WebAssembly module first stores these numbers in memory. Then, it consults the table to find and perform the addition operation. The result, 8, is stored back in memory.

Step 2: Using WebAssembly Tables for Functions

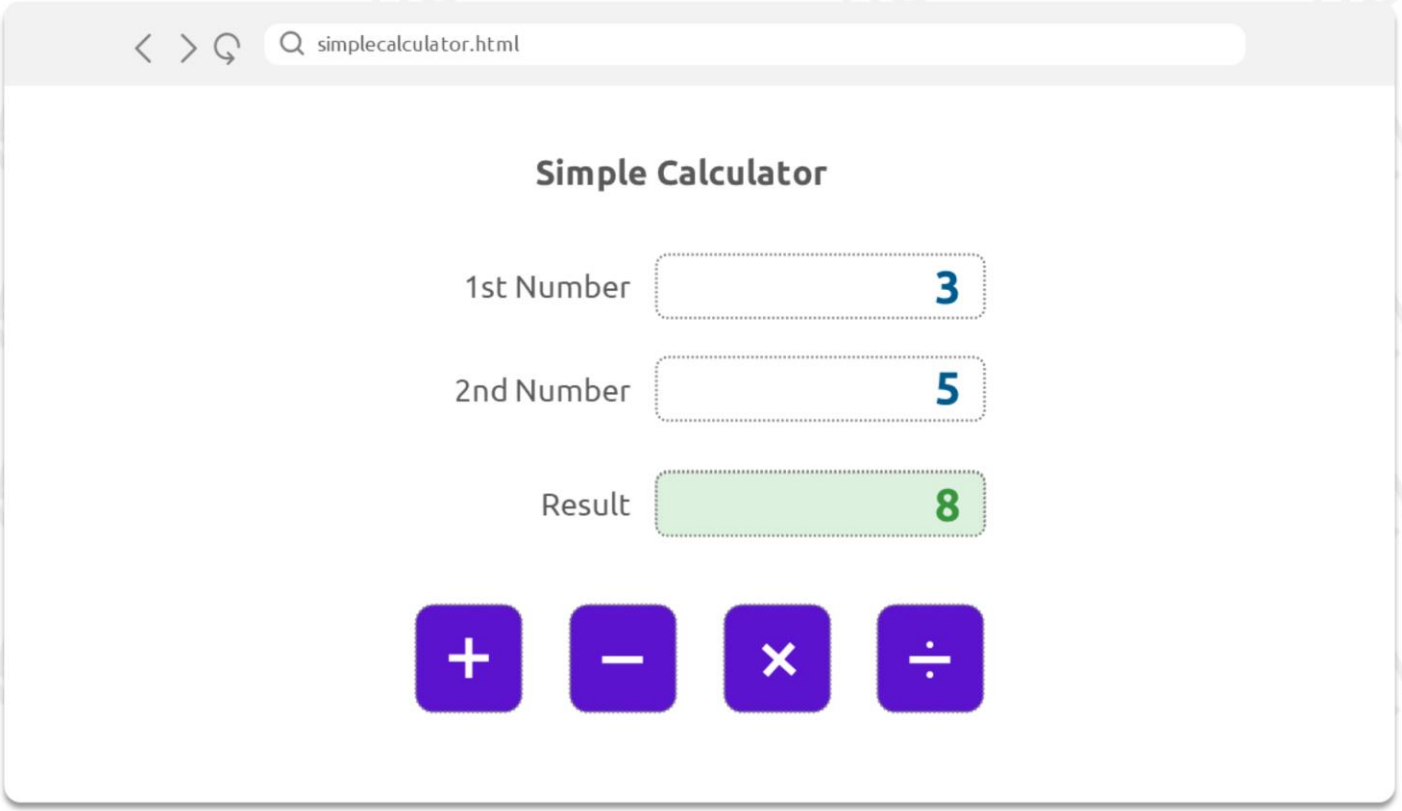


Let's say a user on a website inputs 3 and 5 and selects to add them. Our WebAssembly module first stores these numbers in memory. Then, it consults the table to find and perform the addition operation. The result, 8, is stored back in memory.

Step 2: Using WebAssembly Tables for Functions

Finally, the website can retrieve this result from the WebAssembly memory and display it to the user.

Step 2: Using WebAssembly Tables for Functions



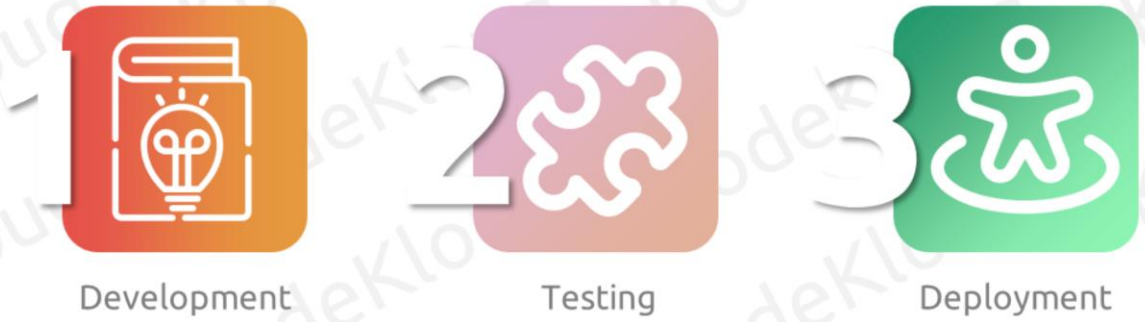
This process shows the collaboration between the memory (storing data and results) and the tables (guiding the program to the right operation).

WASM Tools and Ecosystem

WebAssembly

The world of WebAssembly isn't just about the code and its execution. It's also about the tools and the ecosystem that support, enhance, and streamline the development process. But before we dive into the specifics of these tools and the ecosystem, let's address the fundamental question: Why are they so crucial?

Need for WASM Tools and Ecosystem



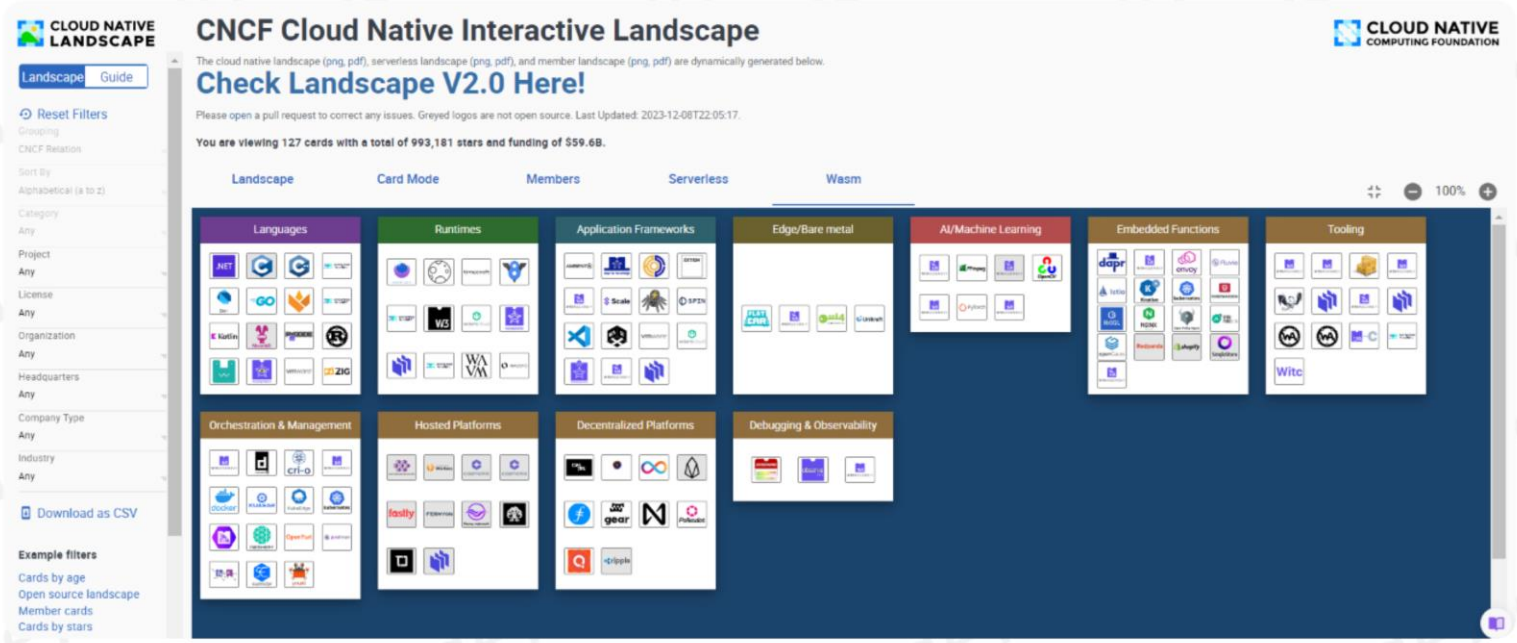
In any craft, having the right tools can make the difference between a smooth process and a challenging one. For WebAssembly, the tools and the surrounding ecosystem simplify the development, testing, and deployment phases, ensuring developers can focus on logic and functionality rather than the intricacies of compilation and optimization.

Innovation: The WASM ecosystem is rapidly evolving, with new tools and libraries emerging regularly. This constant innovation ensures that developers have the best tools at their disposal.

Interoperability: WASM tools are designed to work seamlessly with each other and with other technologies, ensuring a smooth development experience.

Community: A strong community means more collaboration, more shared knowledge, and a better overall developer experience.

The WASM Landscape by CNCF



WebAssembly, or WASM, has seen a surge in its ecosystem, with the Cloud Native Computing Foundation (CNCF) leading the charge by introducing the WASM landscape. This landscape provides a bird's-eye view of the tools, libraries, and projects that are shaping the WASM world. Let's dive into some of its core areas.

Programming Languages

Swift

Compact

When developers embark on creating an application, they kick off with choosing a programming language. WASM's beauty lies in its ability to run applications written in a number of languages. For instance, languages like C, Rust, and Zig can be directly compiled to WASM bytecode, offering swift and compact applications.

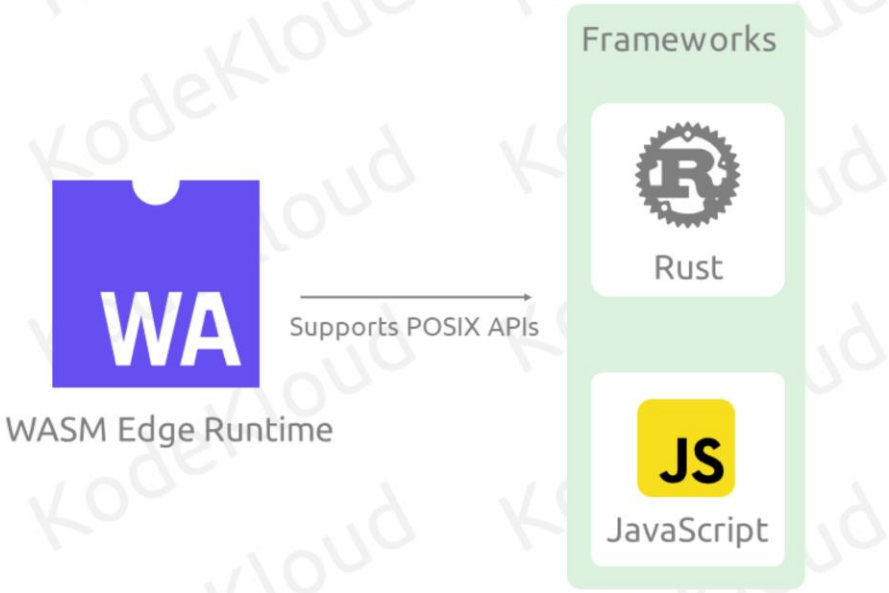
On the other hand, managed languages like Kotlin and Dart rely on the Wasm GC feature, while scripting languages such as JavaScript and Ruby find their way into WASM by compiling their interpreters into it.

There's also a new breed of languages, like Moonbit and Grain, optimized specifically for WASM.

Runtimes

Once you've got your code compiled into WASM bytecode, you'll need a runtime to execute it. Runtimes like WasmEdge, Wasmtime, and WebAssembly Micro Runtime (WAMR) stand out in this domain. They offer the sandboxed safety, speed, and portability that WASM is renowned for.

Application Frameworks



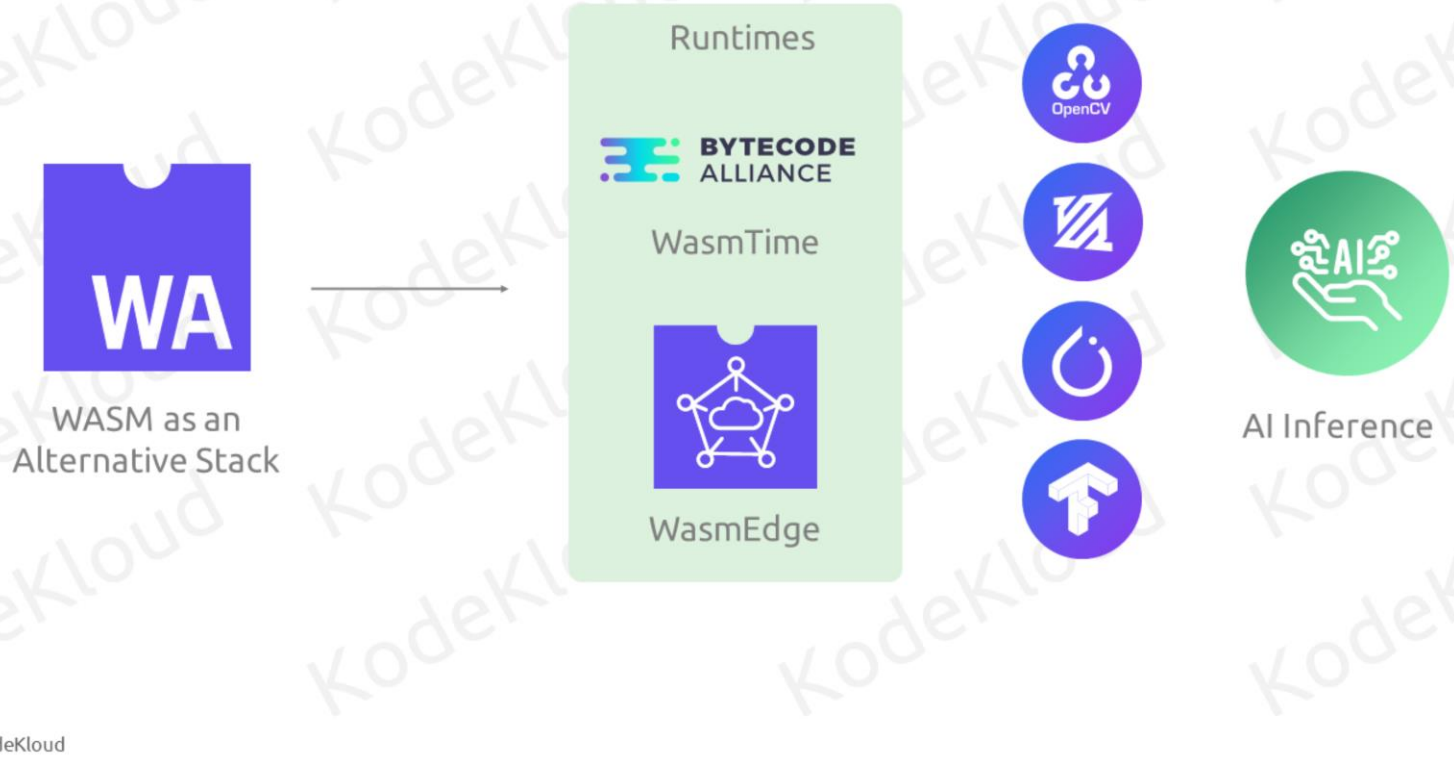
Think of a WASM runtime as an operating system. To make the most of it, developers lean on libraries and frameworks. WasmEdge runtime, for instance, supports advanced POSIX APIs, allowing popular Rust and JS application frameworks to run seamlessly.

Tailored Framework

Then there's Spin, a framework tailored for building WebAssembly microservices, and WasmCloud, which simplifies the development of distributed applications.

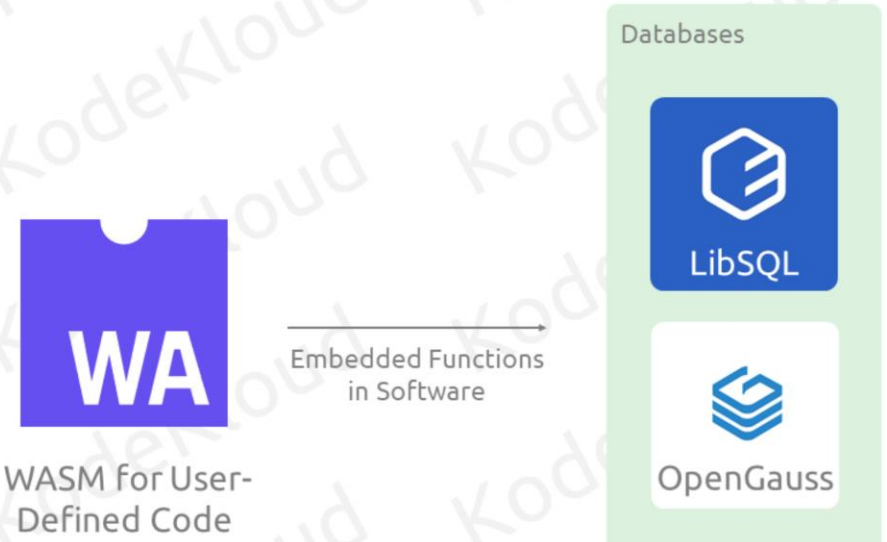
One of WASM's standout features is its cross-platform portability. This means it can run across different operating systems and CPU architectures, even in edge and IoT computing environments. Platforms like Genode, SeL4 RTOS, and Flatcar Container Linux are making waves in this space.

AI Inference



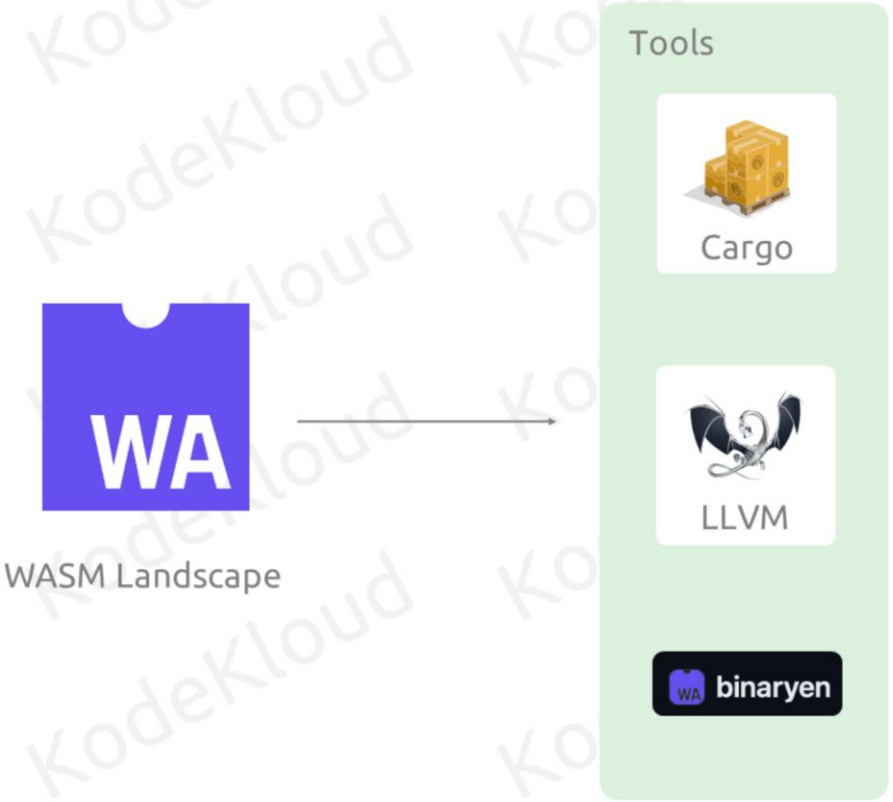
With AI becoming a staple in cloud data centers, WASM is carving a niche for itself as an alternative to traditional stacks. Runtimes like Wasmtime and WasmEdge are integrating with native AI/ML libraries to facilitate AI inference in languages like Rust.

Embedded Function



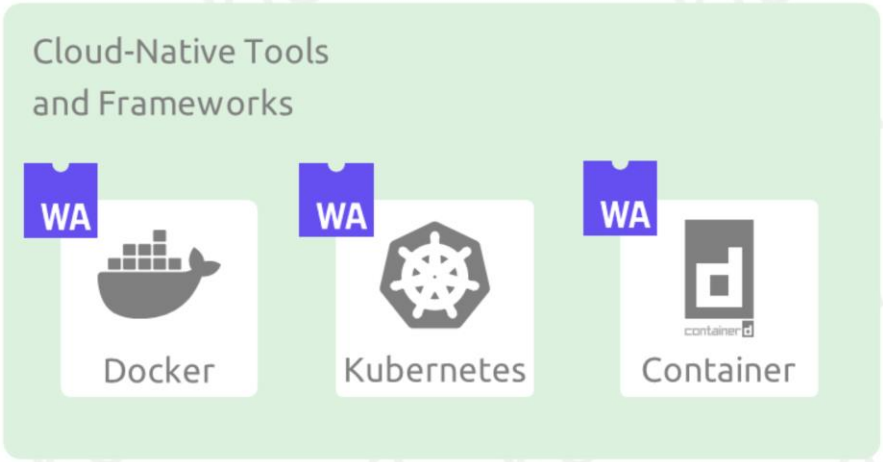
WASM can also be used to execute user-defined code as embedded functions in various software products. For instance, databases like Libsql and OpenGauss have integrated WASM to execute user-defined functions.

Tooling



The maturity of an ecosystem can often be gauged by its tooling. In the WASM landscape, tools like cargo, LLVM, and Binaryen are pivotal for developers building WASM apps.

Application Deployment

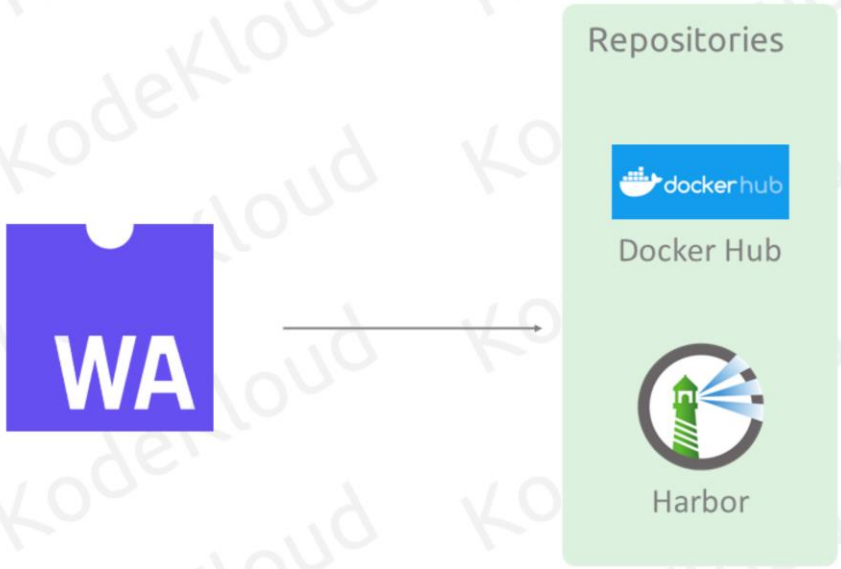


Once a WASM app is ready, it's deployment time. The cloud-native landscape is replete with tools and frameworks, many of which now support WASM. From Docker and Kubernetes to container, deploying and scaling WASM applications has never been easier.

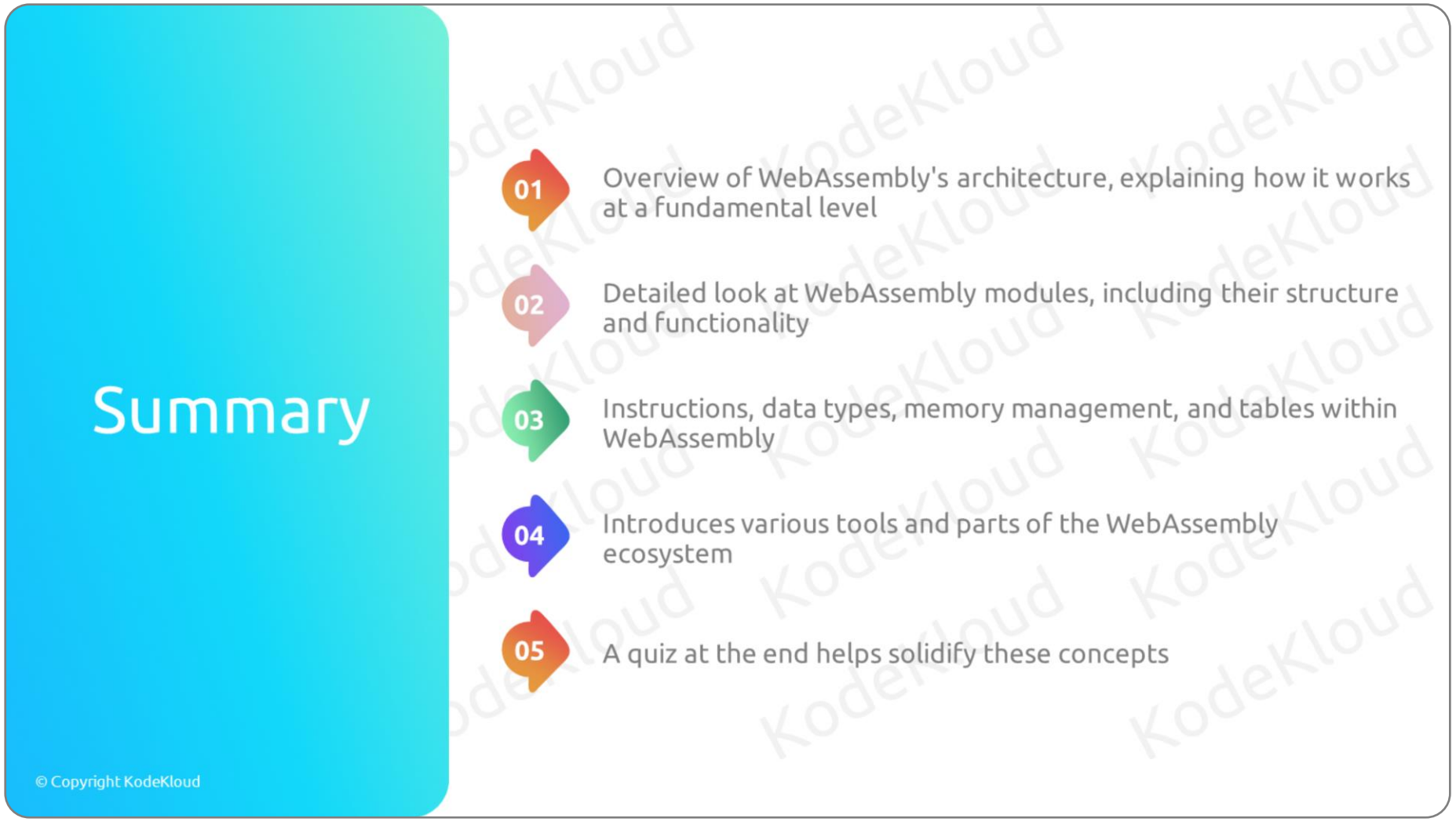
Debugging and Observability

As WASM applications find their way into production, tools for debugging and observability will become indispensable. Currently, tools like WASI logging are making strides in this direction.

Artifacts



Artifact repositories are the backbone of the software supply chain. Repositories like Docker Hub and Harbor are stepping up to store, manage, and track WASM packages.



Here, learners dive into the core concepts of WebAssembly. The section starts with an overview of WebAssembly's architecture, explaining how it works at a fundamental level. This is followed by a detailed look at WebAssembly modules, including their structure and functionality. The section also covers essential topics like instructions, data types, memory management, and tables within WebAssembly. Additionally, it introduces various tools and parts of the WebAssembly ecosystem. A quiz at the end helps solidify these concepts.

KodeKloud

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