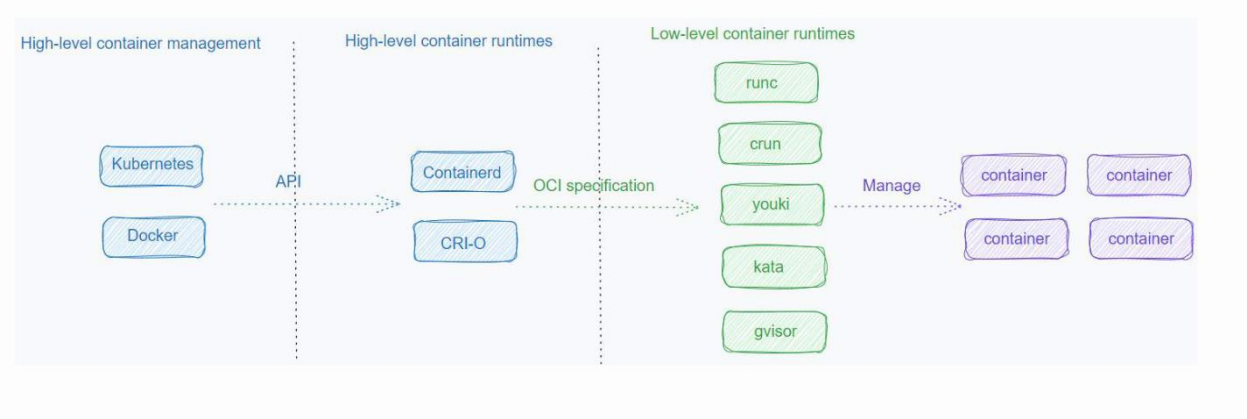
Overview of the execution of wasm

When running a WebAssembly (Wasm) module, the high-level runtime handles loading and communication with the host environment, while the low-level runtime executes the Wasm code and manages security. Docker packages Wasm modules and their environment into containers, making them easy to deploy. Kubernetes then manages these containers, handling things like scaling and networking to ensure everything runs smoothly



When running Wasm modules through high-level container runtimes like CRI-O and contained, there are two main approaches:

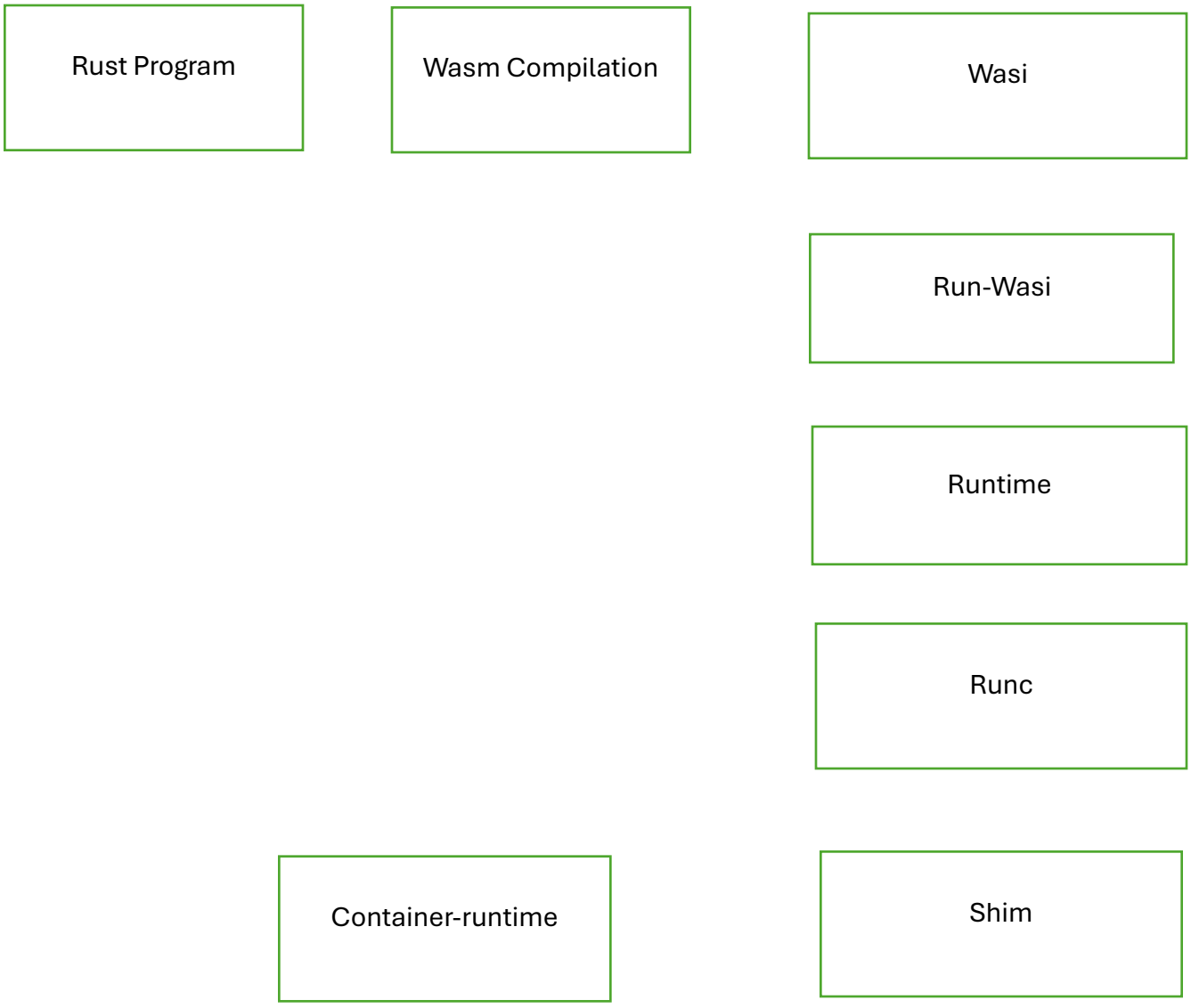
High-Level Runtime with Low-Level Runtime Dependency

1. **High-Level Runtime**:
   - Manages the loading, initialization, and interaction of Wasm modules.
   - Handles communication with the host environment (e.g., I/O operations).
2. **Low-Level Runtime**:
   - Executes Wasm code directly.
   - Ensures security and performance of the Wasm module.
3. **Process Flow**:
   - The high-level runtime invokes the low-level runtime to execute the Wasm module.
   - The low-level runtime manages the execution environment and performs the actual Wasm code execution.

Direct Interaction with Wasm Runtime via `runwasi`

1. **Containerd with `runwasi`**:
   - `runwasi` is a subproject of containerd that facilitates running Wasm modules.
   - It enables the development of a `containerd-wasm-shim`.
2. **Direct Invocation**:
   - The `containerd-wasm-shim` directly interacts with Wasm runtimes like WasmEdge and Wasmtime.
   - This bypasses the traditional low-level runtime, allowing containerd to run Wasm modules more efficiently.
3. **Benefits**:
   - Reduces the invocation path between the high-level runtime and the Wasm execution.
   - Improves overall efficiency by directly utilizing Wasm runtime capabilities

Overall Execution Model of Wasm lookalike as follow



Architecture Description:

1. **Rust Code**:
    - **Source Code**: The initial source code written in Rust.
2. **WASM Compilation**:
    - **Wasm Module**: Converts Rust code into a WebAssembly (WASM) module.
3. **WASI (WebAssembly System Interface)**:
    - **Standardized APIs**: Provides standardized APIs that allow WASM modules to interact with the host system securely.
    - **Portability**: Enables portability of WASM modules across different environments (browsers, cloud, embedded devices).
4. **Run-WASI**:
    - **WASI Execution Tool**: A tool that executes WASM modules with WASI support.
    - **Interaction with Wasmtime**: Facilitates interaction between containerd and Wasmtime, a high-performance WebAssembly runtime.
    - **Runtime Management**: Ensures smoother and quicker execution by managing runtime processes efficiently and eliminating the need for lower-level runtime management.
5. **Runtime Environment**:
    - **Running Wasm Module with WASI**: The environment where the WASM module runs with WASI support.
6. **runc**:
    - **Container Process Management**: Manages the container process.
7. **shim**:
    - **Intermediary Layer**: Acts as an intermediary layer between runc and the container process.
8. **Container Runtime**:
    - **Final Execution Environment**: The environment where the containerized WASM module runs.

9. **Execution Environments**:
    - **Browser**: Web browsers where WASI-enabled applications can run.
    - **Cloud Environment**: Cloud platforms where WASI-enabled applications can be deployed.
    - **Embedded Devices**: Embedded systems where WASI-enabled applications can be executed.

Wasi(Web assembly system interface)

Web Assembly System Interface (WASI) is a new technology that's being introduced to make it easier for software programs to run consistently across different computer systems. Think of it like a common language that allows applications to speak to the underlying hardware of various devices, whether it's a computer, a smartphone, or something else.
Imagine you have a favorite video game. If you want to play it on a different device, like switching from a computer to a tablet, there might be problems. The game is designed to work on your computer, but it might not understand how to talk to the tablet. This is where WASI comes in. WASI acts as a translator, helping the game (or any other software) understand and communicate with the tablet or any device it wants to run on. It creates a common ground, like a universal language, making it possible for the game to run smoothly on different devices without having to be rewritten for each one. WASI (WebAssembly System Interface) is an API that lets WebAssembly code interact with the outside world in a controlled way

WASI is designed to provide a secure standard interface for applications that can be compiled to Wasm from any language, and that may run anywhere—from browsers to clouds to embedded devices. WASI has seen two milestone releases known as **0.1** and **0.2**.

In other words, WebAssembly System Interface, or WASI, is a new family of API's being designed by the Wasmtime project to propose as a standard engine-independent non-Web system-oriented API for WebAssembly. Initially, the focus is on WASI Core, an API module that covers files, networking, and a few other things.

On the Web, it naturally uses the existing Web APIs provided by browsers. However outside of browsers, there's currently no standard set of APIs that WebAssembly programs can be written to. This makes it difficult to create truly portable non-Web WebAssembly programs.
WASI is an initiative to fill this gap, with a clean set of APIs which can be implemented on multiple platforms by multiple engines, and which don't depend on browser functionality.

Capability-Oriented

- **Capability-Based Security**: WASI follows the capability-based security model from CloudABI and Capsicum, where resources like files and network sockets are accessed via file descriptors representing specific capabilities.

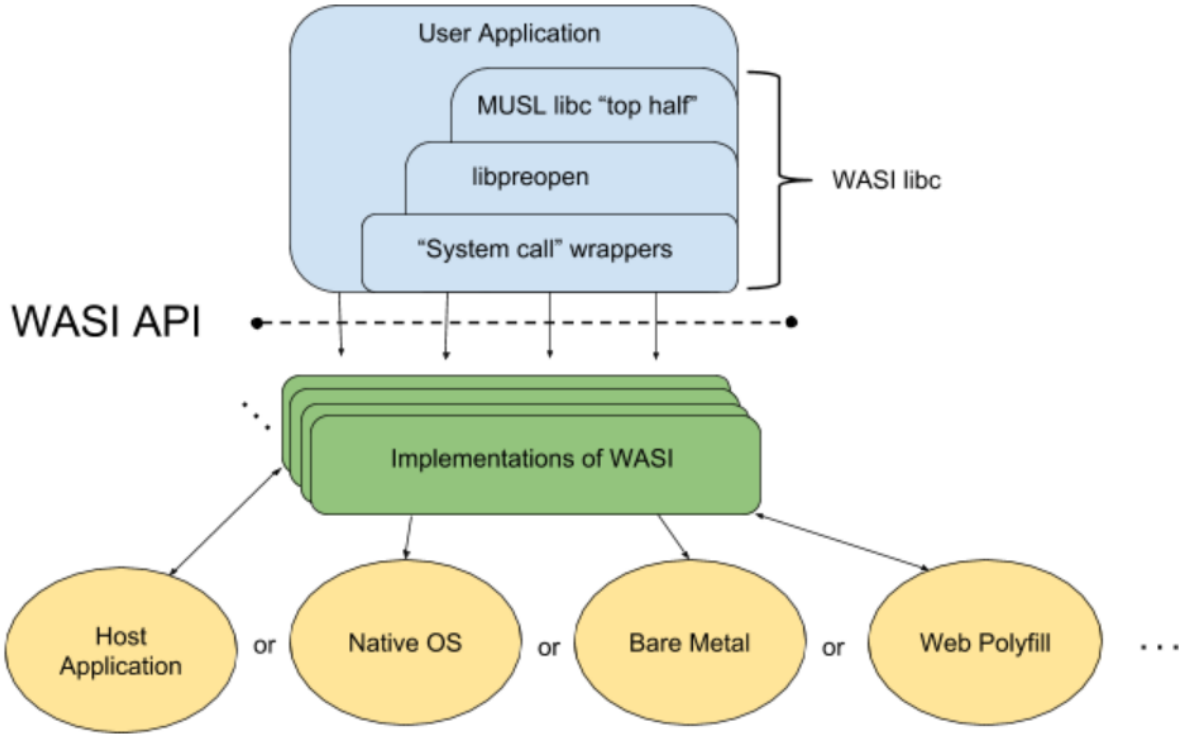
- **Restricted Access**: In WASI, to access external resources (like opening a file), a program must use a file descriptor for the directory containing the file. This restricts access based on granted capabilities.

- **Libpreopen Library**: WASI uses the libpreopen library to manage file access. This library maps file paths to file descriptor indices, allowing existing programs to open files without modifying their code significantly.

- **Ease of Porting**: WASI supports a capability model without needing CloudABI's `program_main` construct, making it easier to port existing applications while maintaining security.

- **Standard I/O Support**: WASI automatically provides file descriptors for standard input and output, and includes a standard library (`musl`-based) for familiar functions like `printf`, aiming for comprehensive libc support

WASI Software Architecture



- **WASI libc**: This is a version of the standard C library tailored for WASI. It provides a normal C library interface but is built on top of layers that manage the specifics of WASI.

- **Libpreopen Layer**: This layer helps manage file access in WASI by mapping file paths to file descriptors. It simplifies file handling for programs by automatically providing the necessary capabilities.

- **System Call Wrapper Layer**: This layer translates function calls from the libc into actual system calls that interact with the WASI implementation. It connects the libc functions with the underlying WASI functionality.

- **WASI Implementation**: This is the core system that handles the actual work of interacting with the operating environment (like native OS resources, JavaScript runtime resources, etc.) based on the system calls it receives.

- **Sysroot**: This is a directory containing the compiled libraries and header files needed to build programs. It provides a standard setup for the compiler to find the libraries and include files it needs.

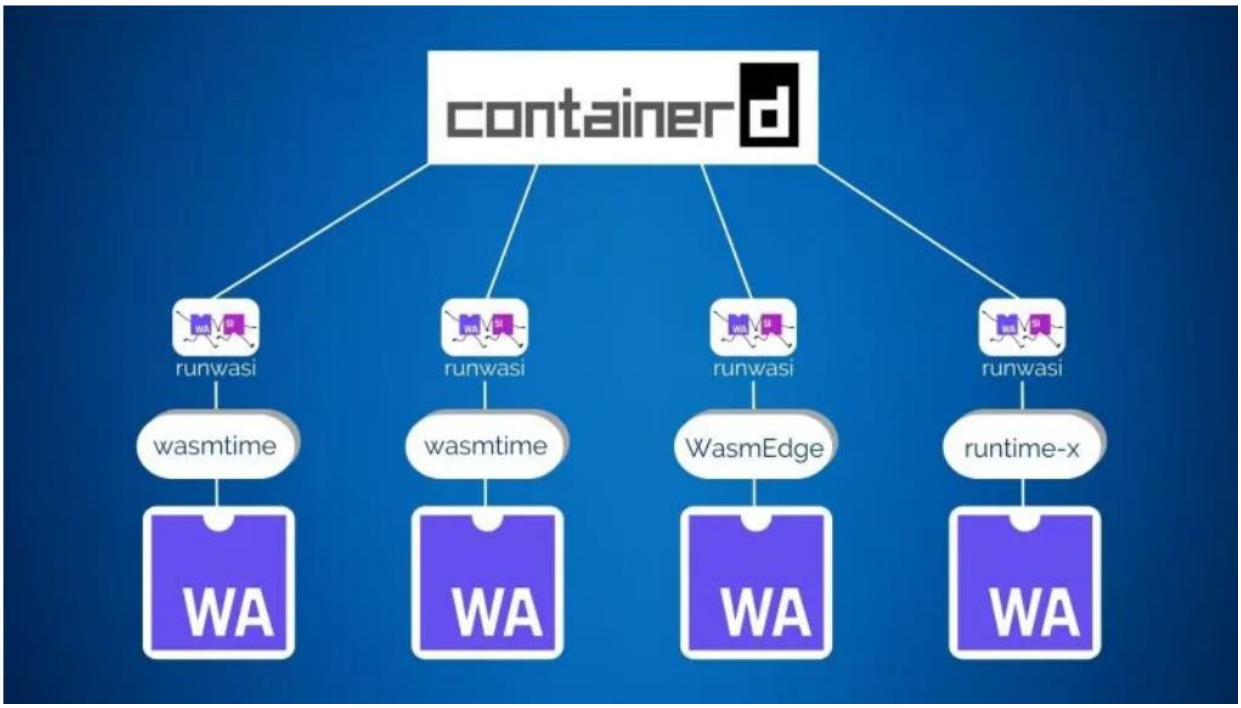
- **LLVM and Clang**: LLVM 8.0 includes a WebAssembly backend that is now stable. Clang is a compiler that can use this backend to compile C/C++ code into WebAssembly. The WASI-enabled sysroot provides the necessary C library functions for this compilation process.

- **Integration**: With the combination of LLVM 8.0, Clang, and the WASI-enabled sysroot, you can compile C and Rust code into WebAssembly modules. These modules can then run in environments that support WASI, like Wasmtime or browsers with WASI polyfills

RunWasi

RunWasi uses WASI to create a secure and standardized environment for running WASM modules. It ensures containerd can easily work with Wasmtime, making the execution of WASM modules simple and efficient. This integration removes the need for complex, low-level runtime management, allowing WASM modules to run smoothly and quickly across different platforms like browsers, cloud services, and embedded devices.

runwasi is a containerd project that let's you swap-out container runtimes for WebAssembly runtimes. It operates as a shim layer between containerd and low-level Wasm runtimes and enables WebAssembly workloads to seamlessly run on Kubernetes clusters.



runwasi is intended to be consumed as a library to be linked to from your own wasm host implementation.

There are two modes of operation supported:

1. "Normal" mode where there is 1 shim process per container or k8s pod.
2. "Shared" mode where there is a single manager service running all shims in process.