THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

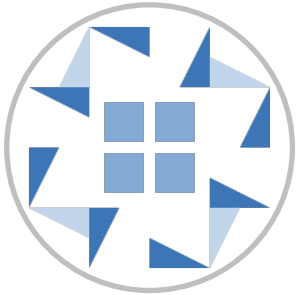
Building a Shared, Persistent Virtual Filesystem for WebAssembly

Ayako Hayasaka
Software Engineer, LY Corporation

Open Source Summit NA 2026

About Me

- Ayako Hayasaka
- Software engineer, LY Corporation in Japan
  - Web backend engineering
- Personal OSS project
  - **kontainer-runtime**: low-level container runtime written in Kotlin/Native
  - **MonakaFS**: host-independent virtual filesystem for WebAssembly



@ternbusty

Agenda

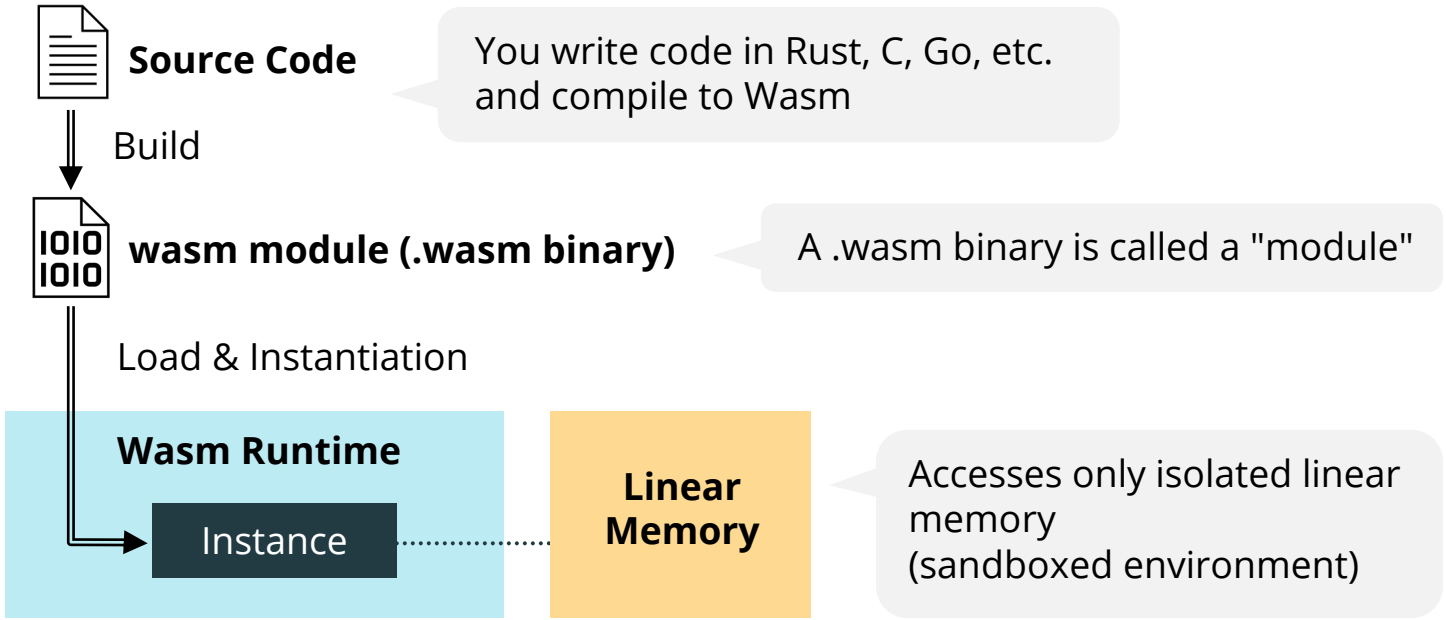
1. WebAssembly basics
2. Why do we need a virtual filesystem for WebAssembly?
3. Three different approaches
    a. Build-time composition (wac plug)
    b. Host-trait implementation (shared in-process)
    c. RPC dynamic attachment (cross-process)
4. Introducing persistence using S3
5. Performance evaluation
6. Which approach to choose?

WebAssembly Basics

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

What is WebAssembly (Wasm)?

A binary instruction format for a stack-based virtual machine originally designed for browsers



Limitations of Wasm Modules

Low-level Interfaces Only

No standard way to pass strings, structs, or complex data

Imports/exports can only pass numbers (i32, i64, f32, f64)

Hard to Compose

Two modules in different languages can't easily talk to each other

Every project invents its own glue code and serialization

Memory Safety

One module can corrupt another’s memory

Modules share linear memory and read/write to each other's memory directly

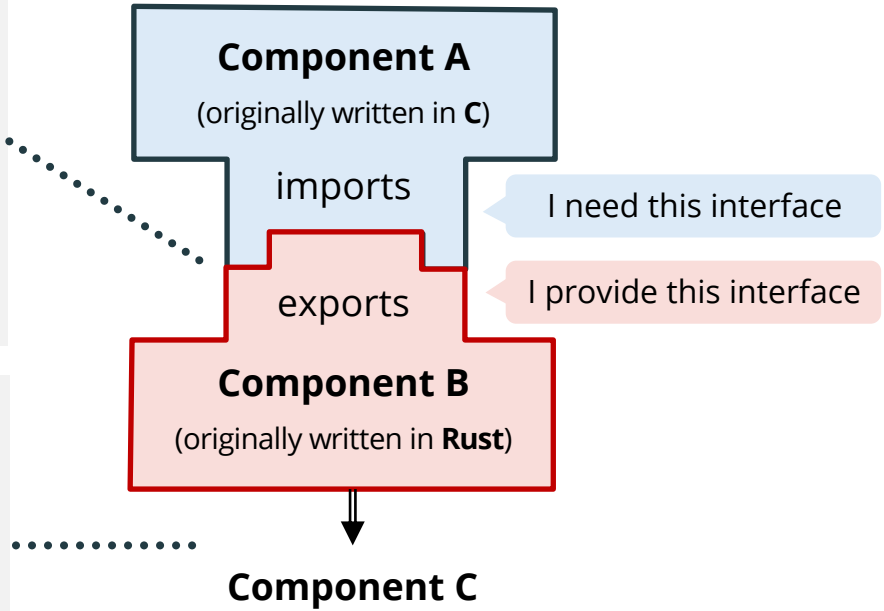
The Component Model

WIT (Wasm Interface Type)

- The interface definition language
- Rich types: strings, lists, ...
- Components can import and export WIT interfaces

Composition

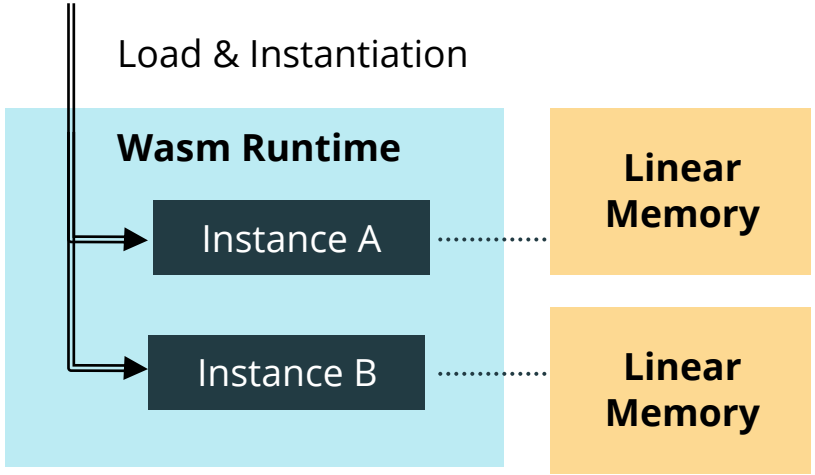
- **wac plug**: Plug exports into imports, produce a new component



The Component Model

Memory Safety

- Each component has its own isolated linear memory and passes data through WIT interface
- Canonical ABI: lifting / lowering copies data across boundaries



Why Do We Need a Virtual Filesystem for WebAssembly?

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

Wasm Beyond the Browser

- Now expanding to server-side, edge computing, IoT, and more
- Why use Wasm outside the browser?
  - **Lightweight & fast**: microsecond-order cold starts
    - Frameworks like Fermyon Spin appeared
  - **Sandboxed**: memory isolation from the host (linear memory)
  - **Portable**: same binary runs anywhere a Wasm runtime exists

WASI: Accessing Host Resources

- WebAssembly itself has no I/O, it's pure computation
- WASI (WebAssembly System Interface) bridges that gap
  - Preview 1 (stable), Preview 2 (2024, Component Model based)
- Filesystem access via **preopen**:
  - Host grants access to specific directories
  - Wasm app directly uses the host's filesystem

Problems with preopen

- **Security risks**: Weakens the Wasm sandbox model
  - CVE-2023-51661: runtime bug allowed unauthorized host FS access
  - The whole point of Wasm sandboxing is lost if apps touch host files
- **Portability issues**:
  - Path separator differences (Windows vs Unix)
  - Environment-dependent behavior breaks "write once, run anywhere"

We need a virtual filesystem independent of the host filesystem!

Existing Solutions and Their Limitations

	Description	WASI	Read	Write	Persistence	Multi-app Sharing	Note
wasi-vfs	A virtual filesystem layer for WASI	**Preview 1**	O	**X**	**X**	**X**	**Hard to use from languages like Rust** because of wasi-libc dependency
wasi-virt	Virtualization Component Generator for WASI Preview 2 by Bytecode Alliance	Preview 2	O	**X**	**X**	**X**	

Neither meets all of our requirements

Existing Solutions and Their Limitations

	Description	WASI	Read	Write	Persistence	Multi-app Sharing
WasmFs	emscripten's VFS	- (**Browser Only**)	O	O	O	X
wasmer virtual-fs	VFS for wasmer (wasm runtime)	**Not Standard** (WASIX)	O	O	X	X
Cloudflare Workers	/tmp per request	**Preview 1**	O	O	**X**	**X**
Fermyon Spin	**VFS is not provided**, host mount or use of KV Store are recommended					
Fastly Compute	**No filesystem access**, use of KV store is recommended					

Platform / Runtime dependent solutions are also limited

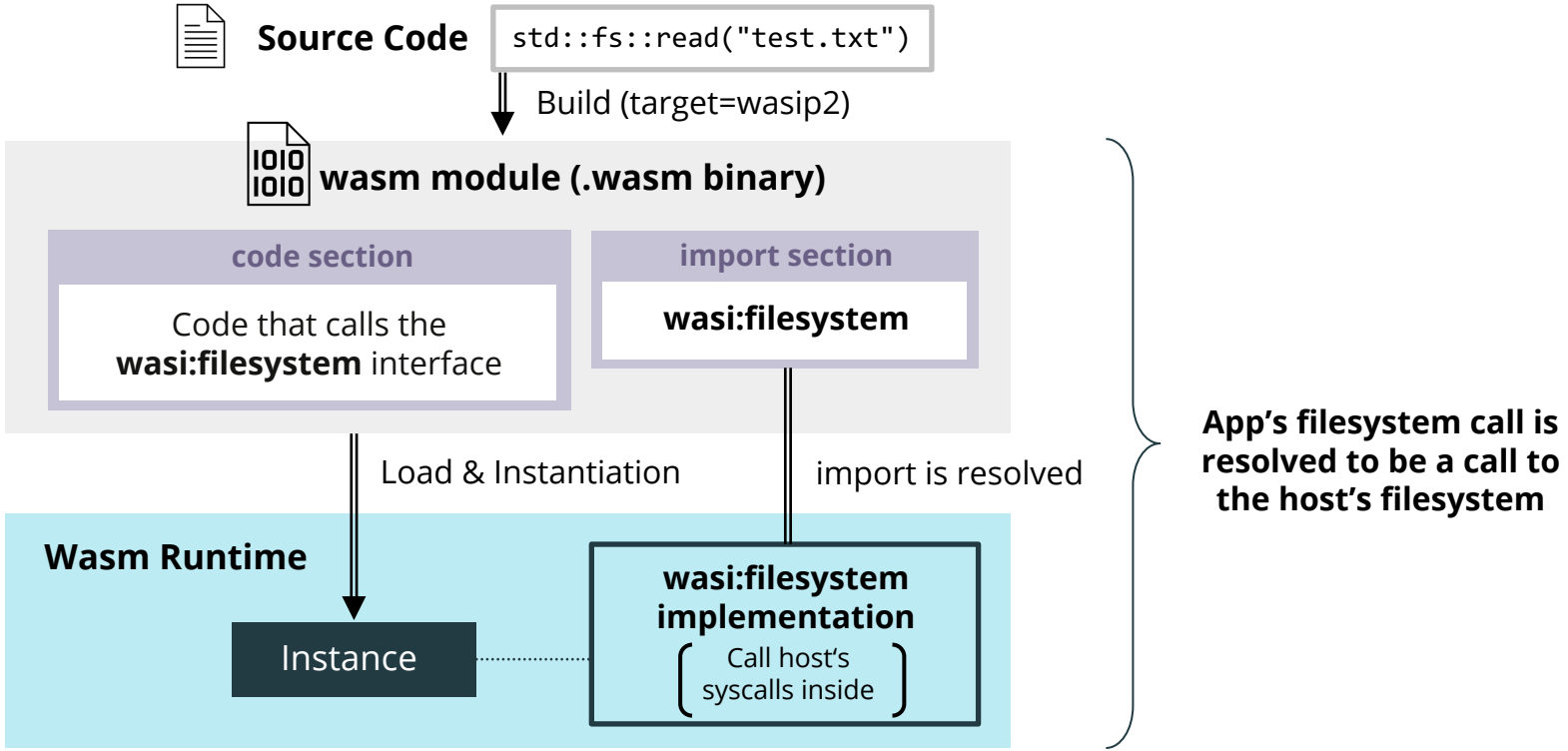
Our Goals

Build a host-independent Virtual File System (VFS) for WebAssembly:

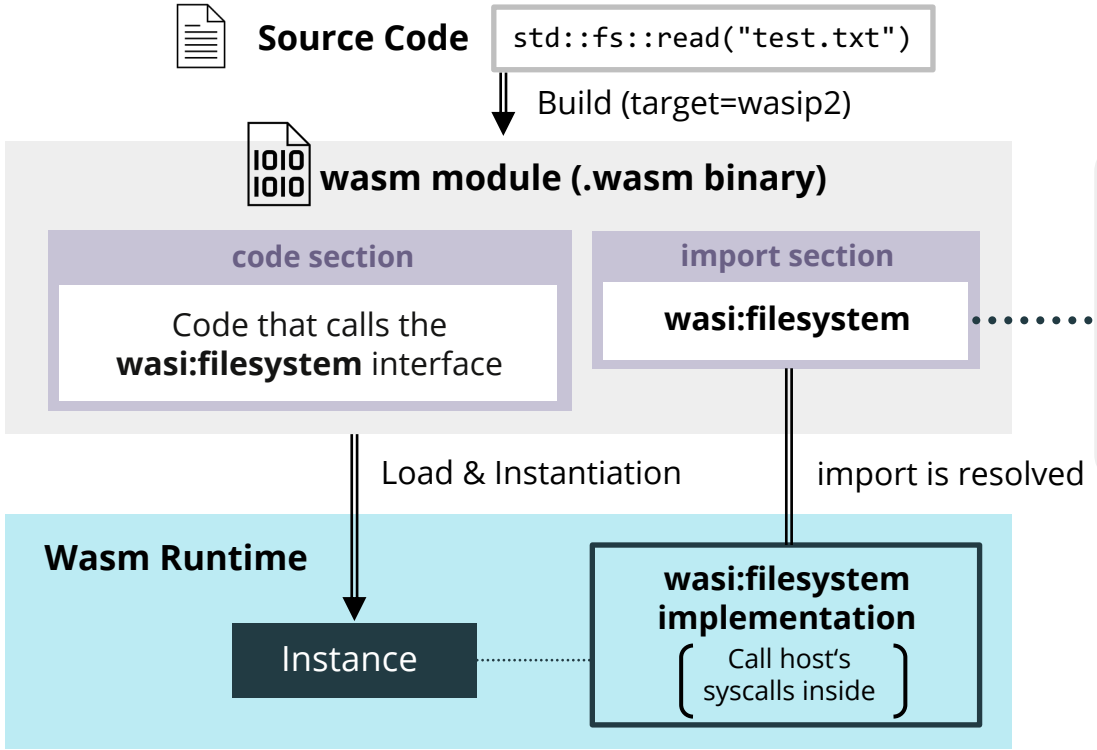
1. **Logical isolation from host OS filesystem**
    - No preopen, no host FS dependencies
2. **Existing apps work without modification**
    - Applications can use standard I/O such as `fopen` or `std::fs`
3. **Flexible deployment options**
    - Build-time composition, multi-app sharing, RPC, S3 persistence

Three Different Approaches

File System Access Using WASI



File System Access Using WASI



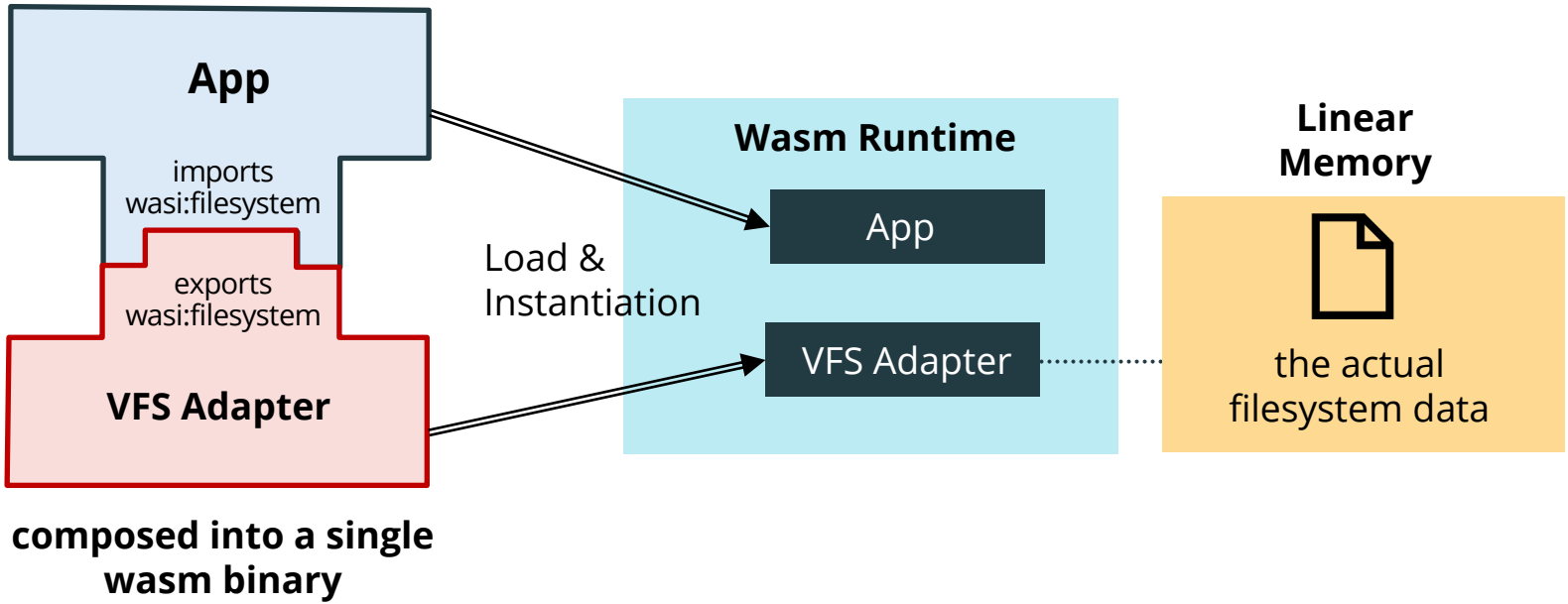
Replacing the FS Implementation

By providing a custom implementation that satisfies the **wasi:filesystem** interface definition, filesystem calls are directed to our implementation instead of the host's.

Three Different Approaches:
1. Build-time composition

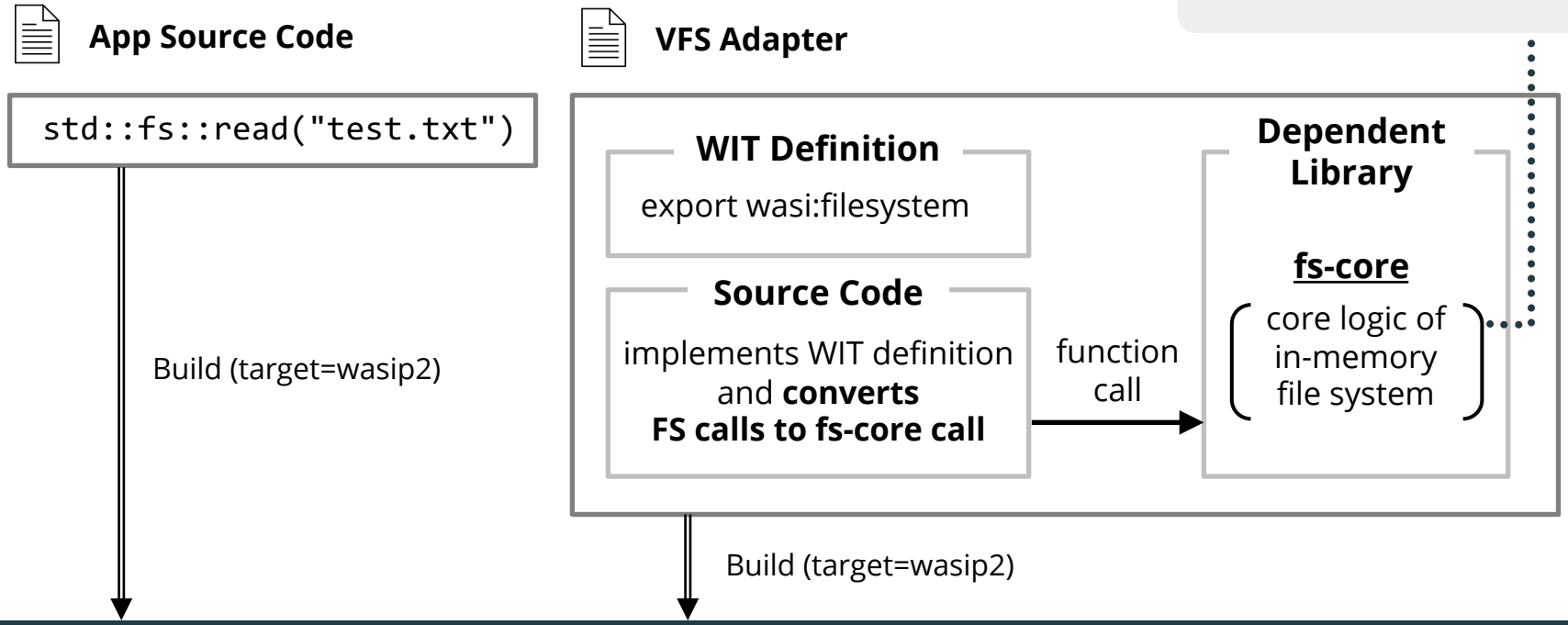
Approach 1: Build-Time Composition

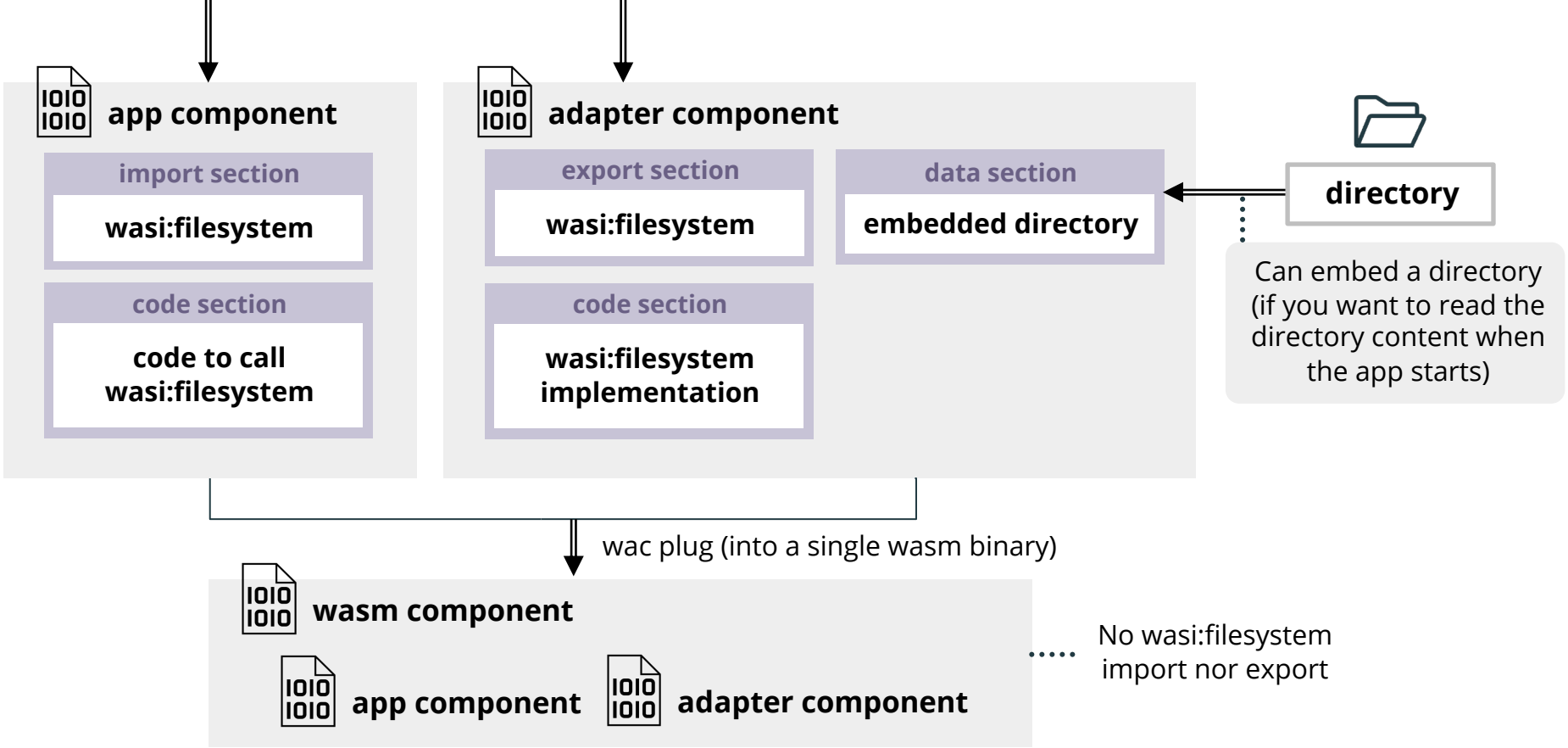
wasi-virt-like architecture



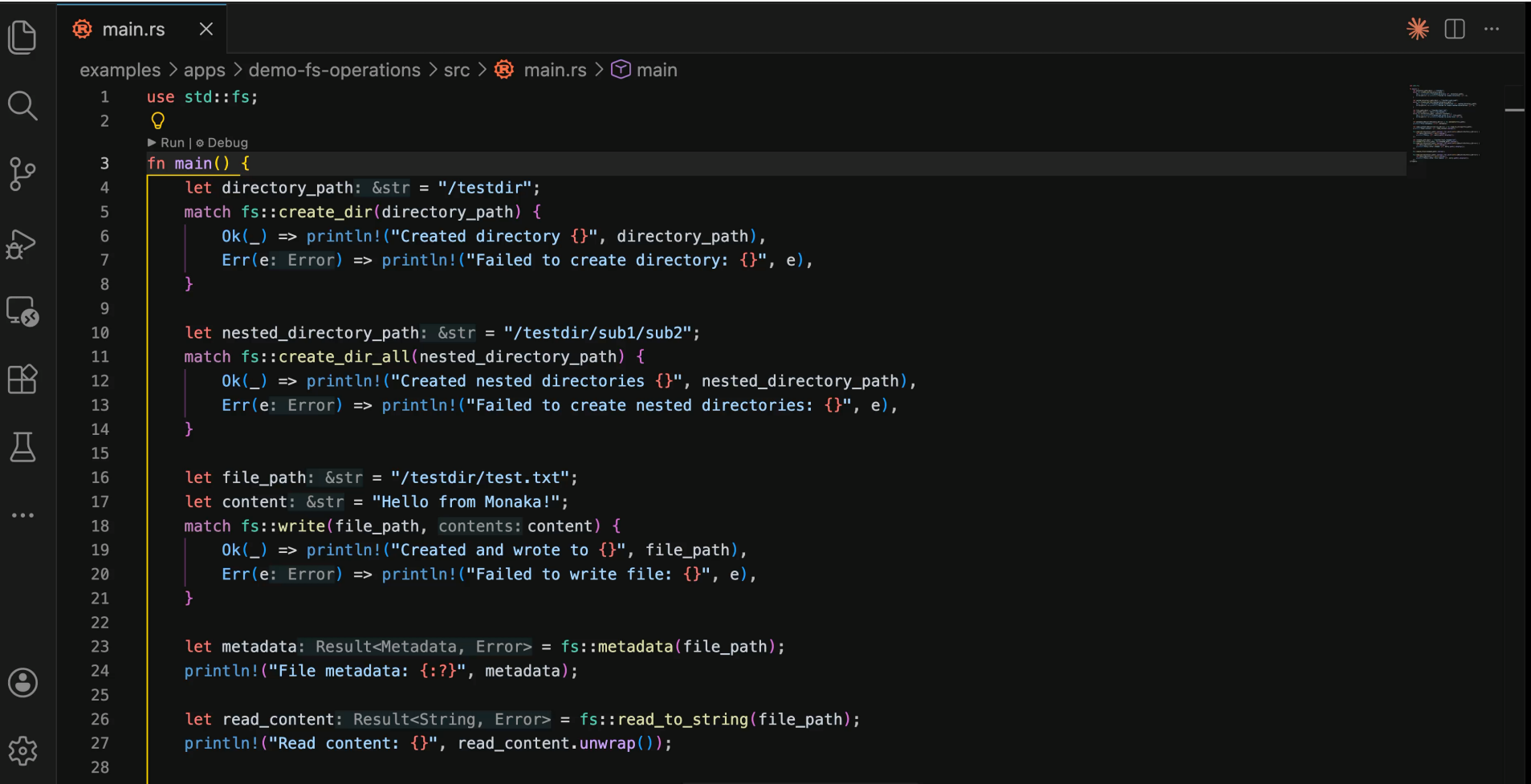
Approach 1: Build-Time Composition

Provide in-memory VFS function such as Inode management, block assignment

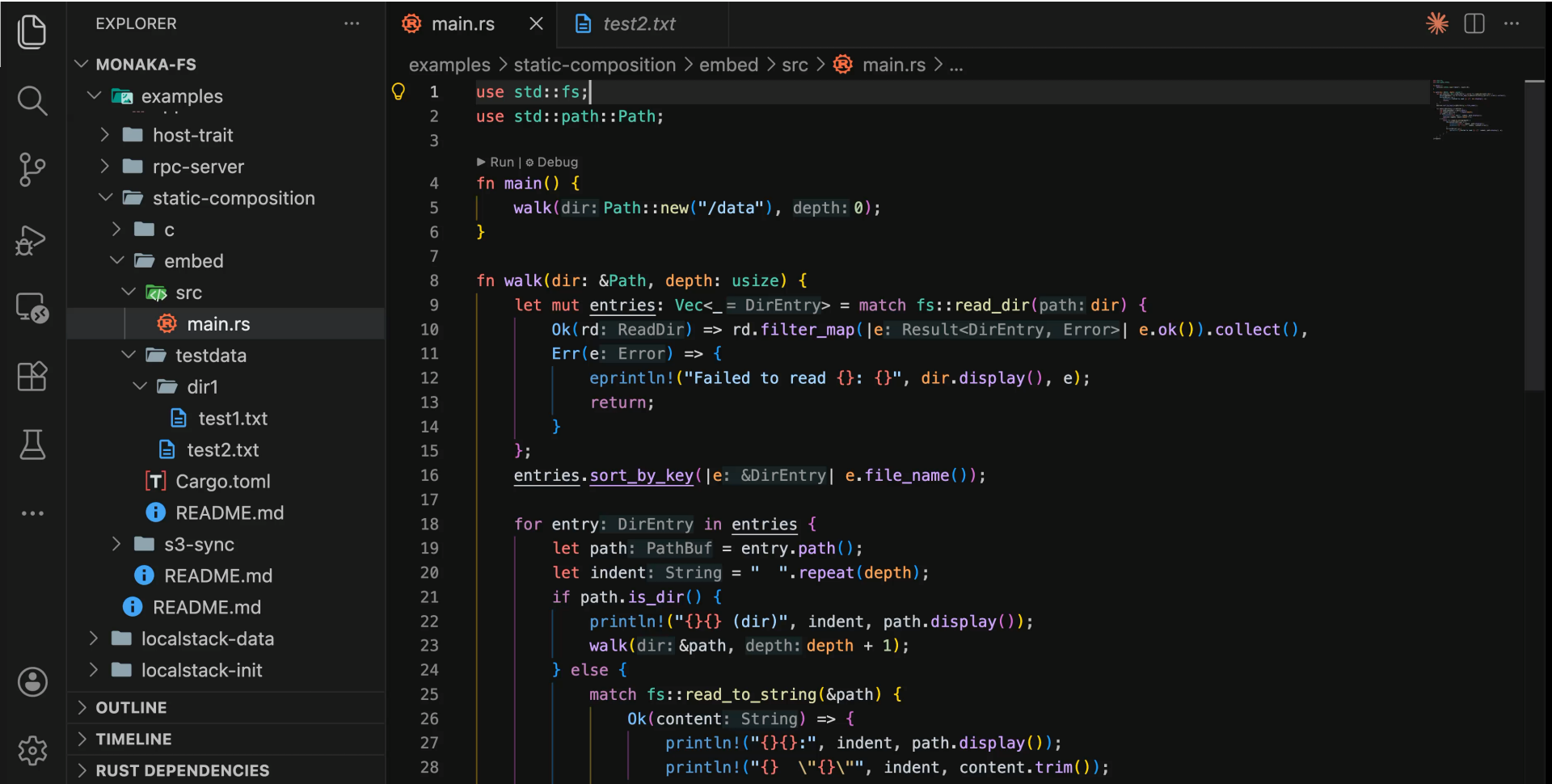



app component
import section
wasi:filesystem
code section
code to call wasi:filesystem
adapter component
export section
wasi:filesystem
data section
embedded directory
code section
wasi:filesystem implementation
directory
Can embed a directory (if you want to read the directory content when the app starts)
wac plug (into a single wasm binary)
wasm component
app component
adapter component
No wasi:filesystem import nor export

Approach 1: Basic File Operation Demo



Approach 1: Directory Embed Demo



Approach 1: Use Cases & Limitations

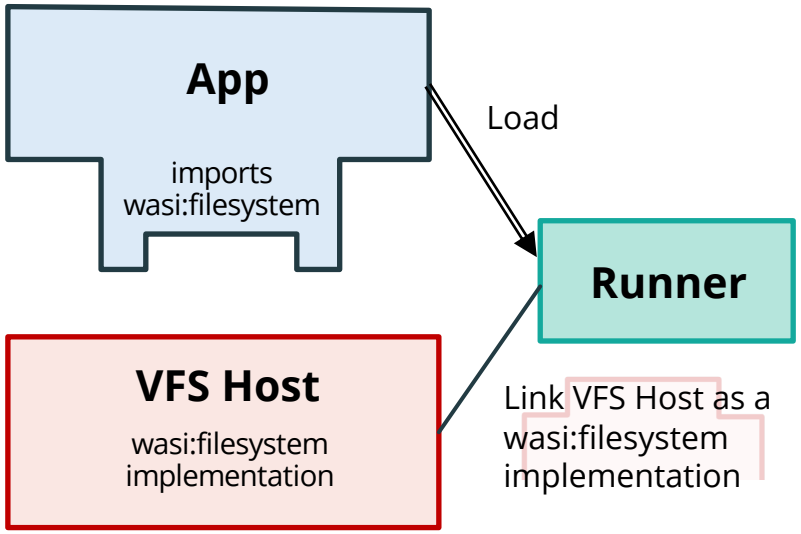
- Use Cases
  - Bundle config files into the binary
    - Embed directories at build time via CLI
  - Temporary working area
    - Image processing pipeline, etc.
- Limitations
  - No persistence, data lost on exit
  - Cannot share FS between multiple apps

What if we need multi-app sharing?

Three Different Approaches: 2. Host-trait implementation (shared in-process)

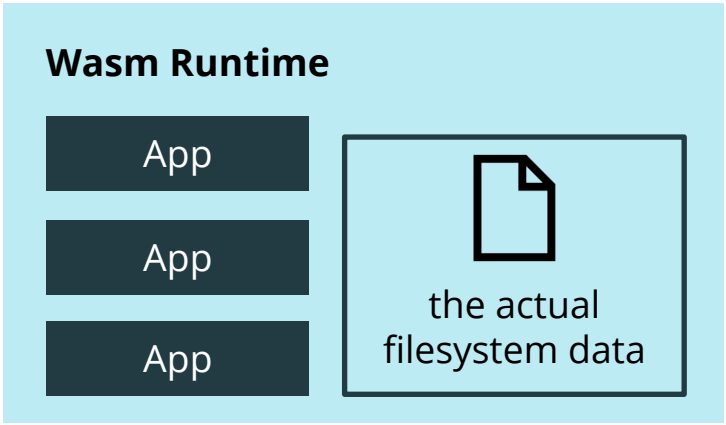
Approach 2: Host Trait Implementation

Overview



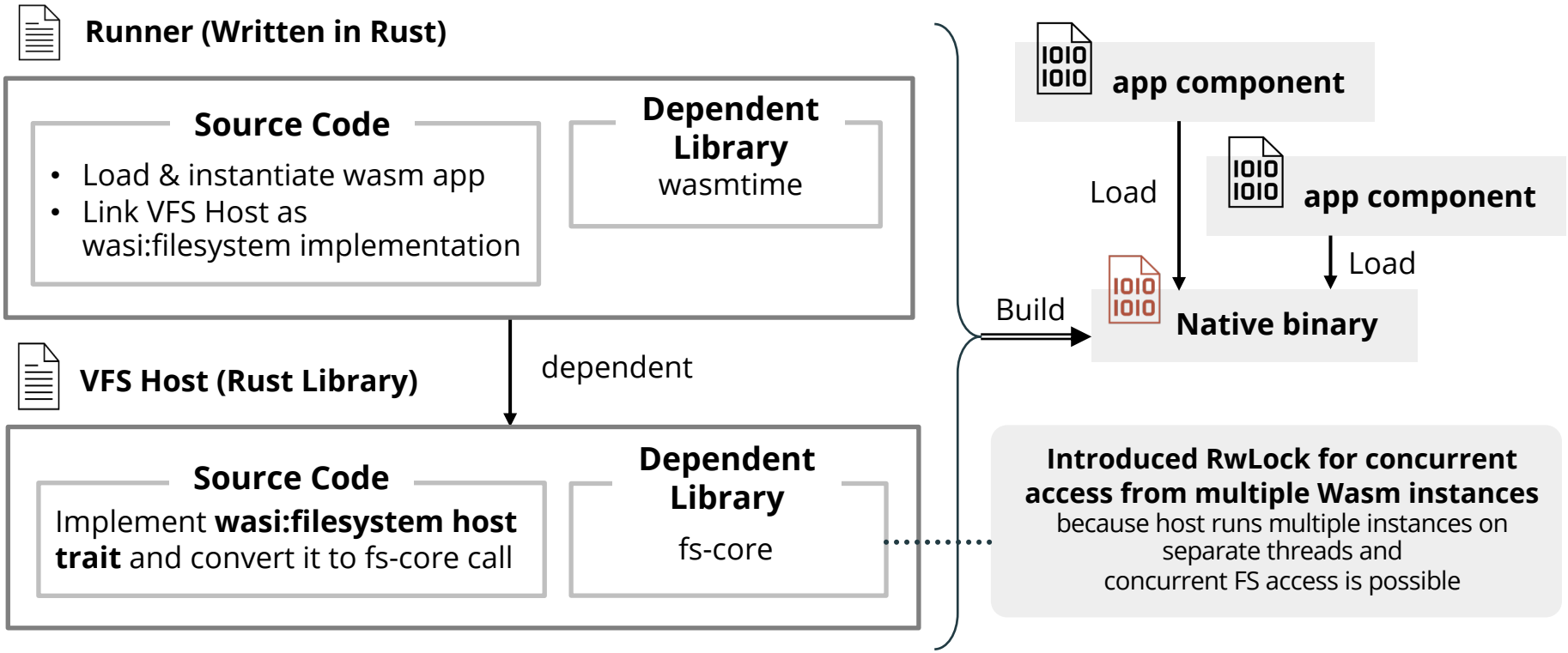
wasi:filesystem implementation is provided from host

When Running...

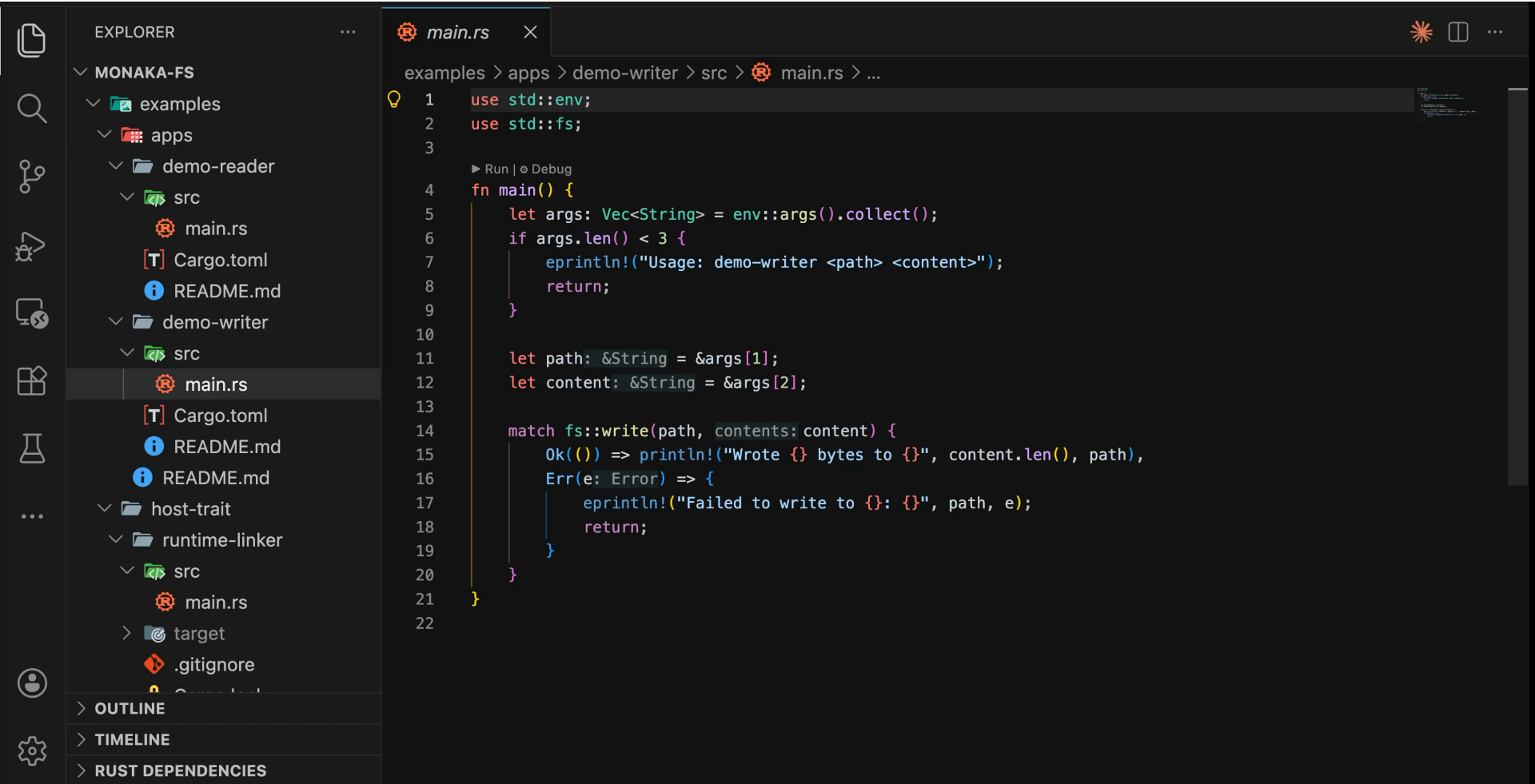


The actual filesystem data is in the host memory (not linear memory) and can be shared with multiple apps within the same process

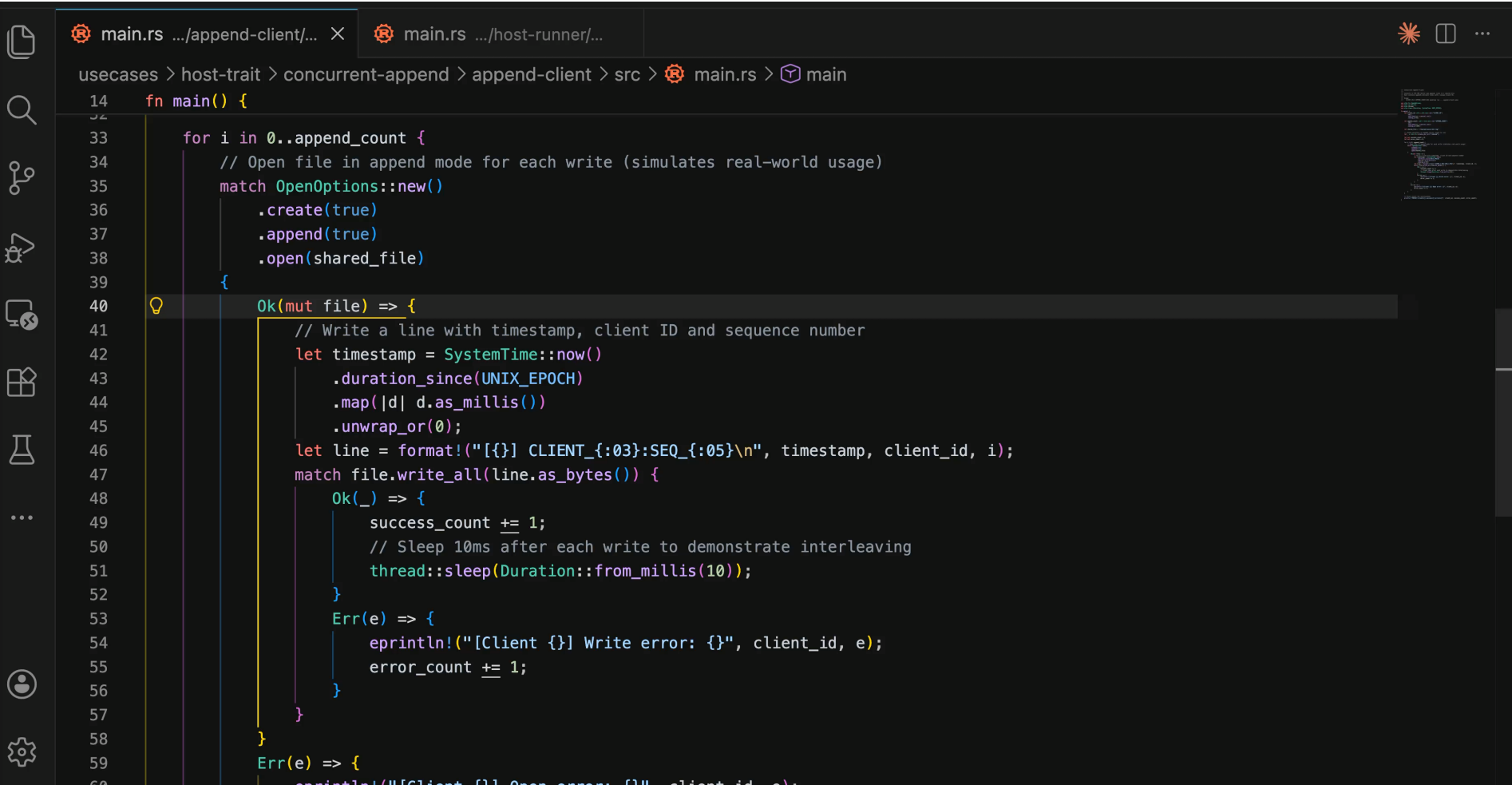
Approach 2: Host Trait Implementation



Approach 2: Filesystem Sharing Demo



Approach 2: Concurrent Writing



Approach 2: Cache Server

main.rs .../http-server/... | main.rs .../request-handler/...

usecases > host-trait > http-cache > http-server > src > main.rs > main

```
87  async fn main() -> Result<()> {
101         .join(path: "request-handler/target/wasm32-wasip2/debug/http-cache-handler.wasm");
102
103     // Load handler component
104     let handler_component: Component =
105         Component::from_file(&engine, file: &handler_path).context("Failed to load handler component")?;
106
107     // Create initial VFS state and extract shared VFS (no WASM adapter needed)
108     println!("Initializing VFS...");
109     let initial_vfs_state: VfsHostState =
110         vfs_host::VfsHostState::new().context("Failed to create VfsHostState")?;
111     let shared_vfs: Arc<Fs> = initial_vfs_state.get_shared_vfs();
112
113     // Create app state
114     let state: Arc<AppState> = Arc::new(data: AppState {
115         engine,
116         handler_component,
117         shared_vfs,
118     });
119
120     // Create router
121     let app: Router = Router::new() Router<Arc<AppState>>
122         .route(path: "/api/*path", method_router: get(handler: handle_api_request)) Router<Arc<AppSt…
123         .route(path: "/health", method_router: get(handler: || async { "OK" })) Router<Arc<AppSt…
124         .with_state(state);
125
126     // Start server
127     let addr: &str = "0.0.0.0:8080";
128     println!();
```

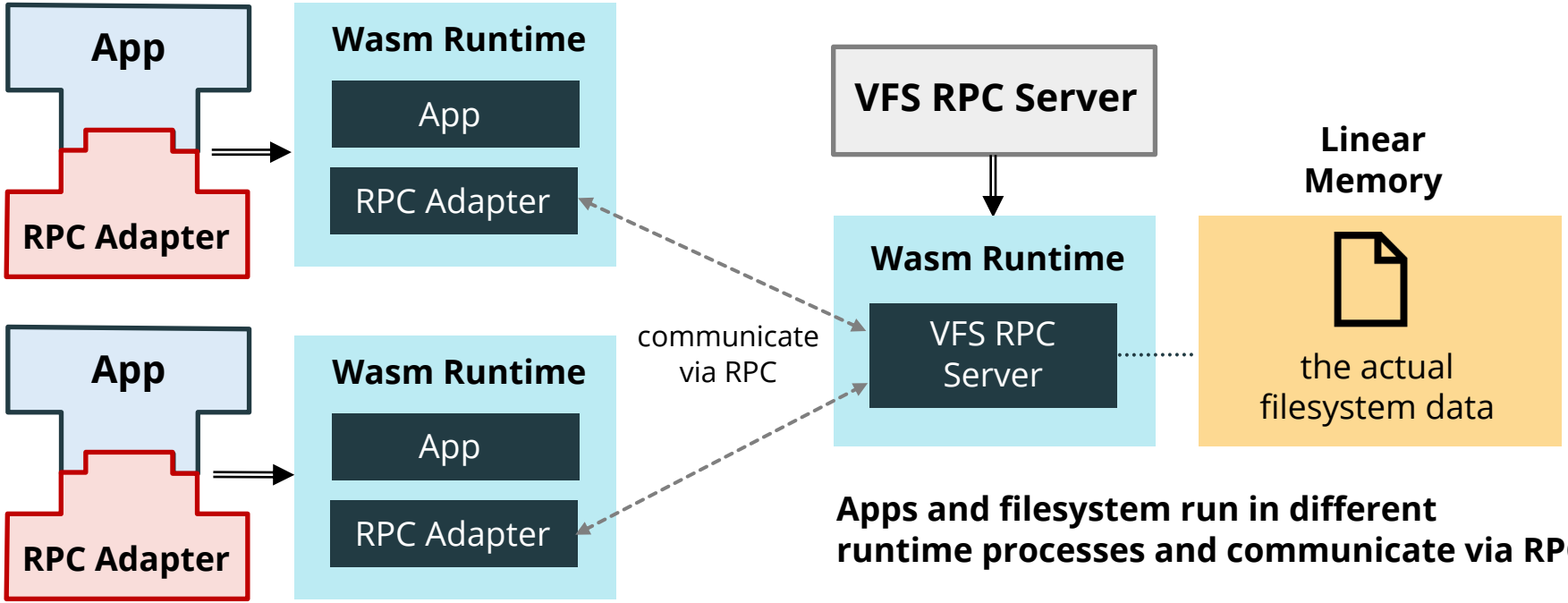
Approach 2: Use Cases & Limitations

- Use Cases
  - Edge device data pipelines
    - Sensor -> temp file -> processing module
  - Cache server (Fermyon Spin style)
    - Per-request Wasm instances share FS
    - FS as cross-request cache
- Limitations
  - No persistence, data lost on exit
  - Must distribute a native binary
  - **Limited to sharing within one host process**

What about cross-process sharing?

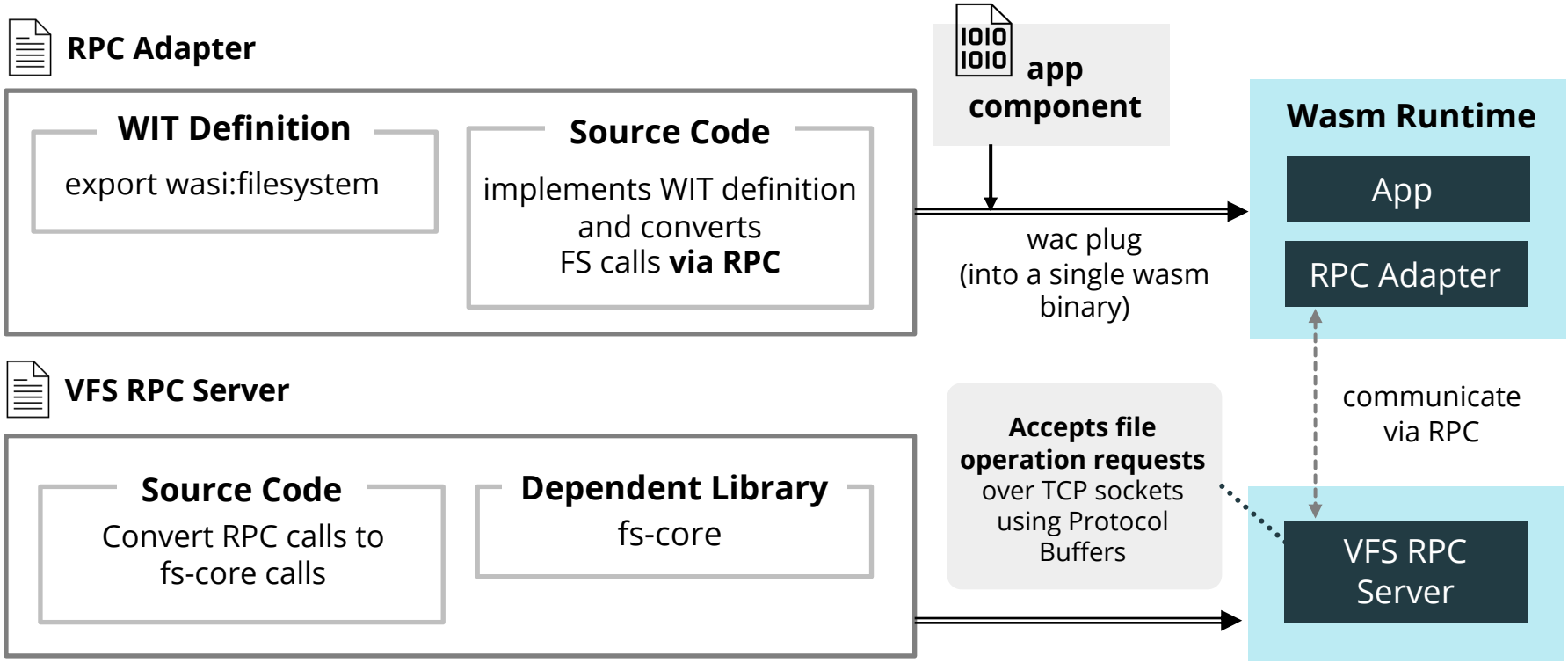
Three Different Approaches: 3. RPC dynamic attachment (cross-process)

Approach 3: RPC Dynamic Attachment



Apps and filesystem run in different runtime processes and communicate via RPC

Approach 3: RPC Dynamic Attachment



Approach 3: Filesystem Sharing Demo

~/workspace/public/monaka-fs main ± 0

```
monaka compose --rpc target/wasm32-wasip2/debug/demo-writer.wasm -o /tmp/rpc-writer.wasm
monaka compose --rpc target/wasm32-wasip2/debug/demo-reader.wasm -o /tmp/rpc-reader.wasm
```

Approach 3: Use Cases & Limitations

- Use Cases
  - CI/CD build cache sharing across pipeline stages
  - Dynamic scaling: new workers connect to existing FS
  - Processes with different lifecycles sharing data
- Limitations
  - Network latency on every FS operation
  - No persistence, data lost on exit

How can we add persistence?

Introducing persistence using S3

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

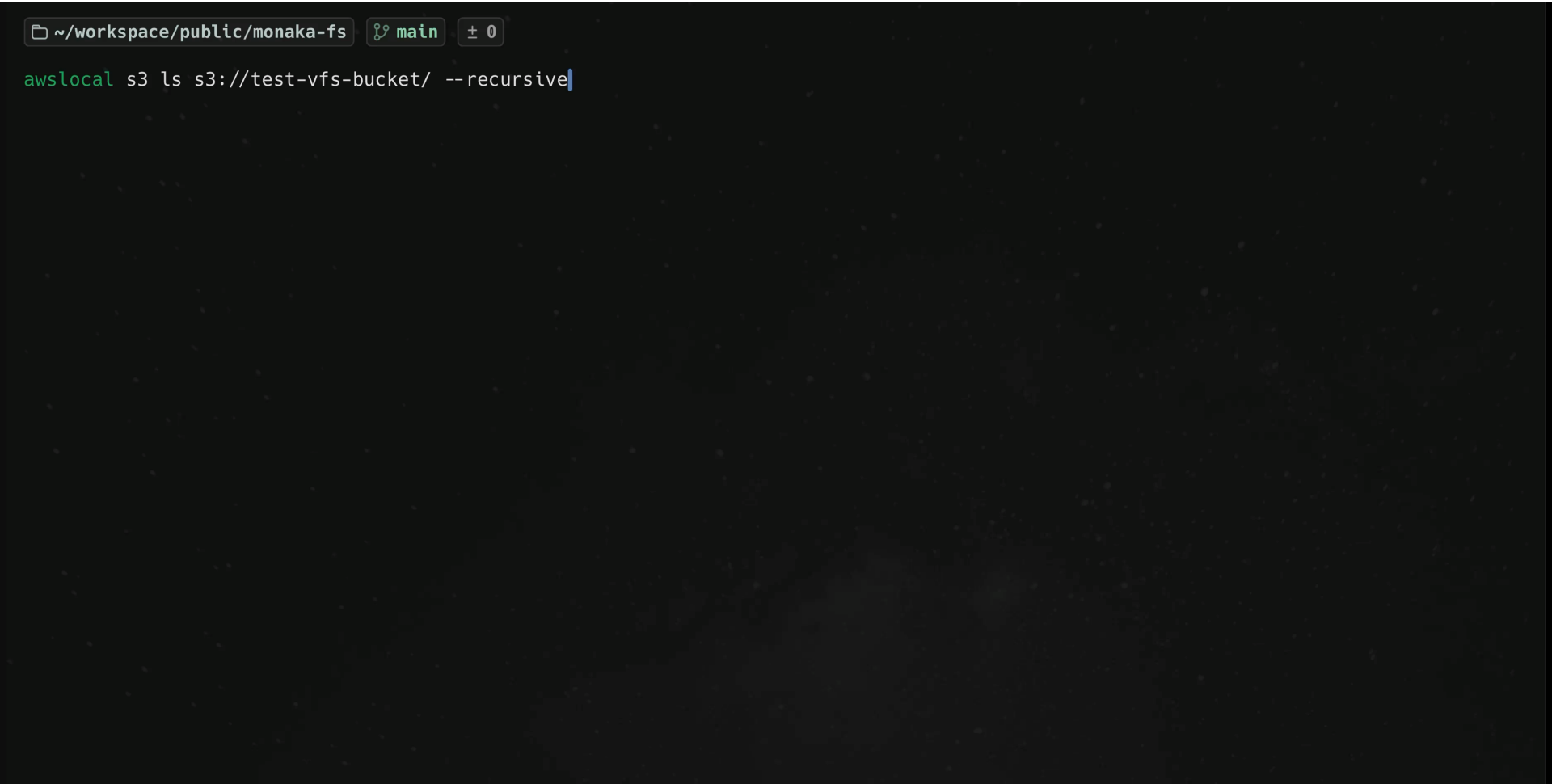
Adding persistence using S3

- Problem: all 3 approaches are in-memory, data lost on exit
- Solution: **sync the VFS with Amazon S3-compatible storage**
  - Similar concept to **s3fs-fuse** on Linux
  - But no FUSE, no root privileges, no kernel dependencies
  - Works on any platform
- Can be added to any of the 3 approaches:
  - Build-Time Composition: add to VFS Adapter
  - Host-trait implementation: add to VFS Host
  - RPC dynamic attachment: add to VFS RPC Server

S3 Sync Strategies

- Read path:
  - Read-through: every read hits S3
  - Memory-cache: serve reads from memory
- Write path:
  - Write-through: upload on every write
  - Async: queue writes, upload in background
- Background sync:
  - Cold-start restore: load S3 state on startup
  - ETag-based polling: detect remote changes

S3 Sync Demo (VFS -> S3)



S3 Sync Demo (cold start & restore)

```
~/workspace/public/monaka-fs git:(main)  (0.234s)
wasmtime run -S inherit-network=y /tmp/rpc-writer.wasm "messages_by_writer.txt" "This is a message from Writer"
Wrote 29 bytes to messages_by_writer.txt

~/workspace/public/monaka-fs git:(main)  (2.694s)
awslocal s3 ls s3://test-vfs-bucket/ --recursive
2026-04-13 23:47:57         29 vfs/files/messages_by_writer.txt

~/workspace/public/monaka-fs git:(main)  (0.481s)
awslocal s3 cp "s3://test-vfs-bucket/vfs/files/messages_by_writer.txt" -
This is a message from Writer%

~/workspace/public/monaka-fs  main  ± 0
```

S3 Sync Demo (S3 -> VFS)

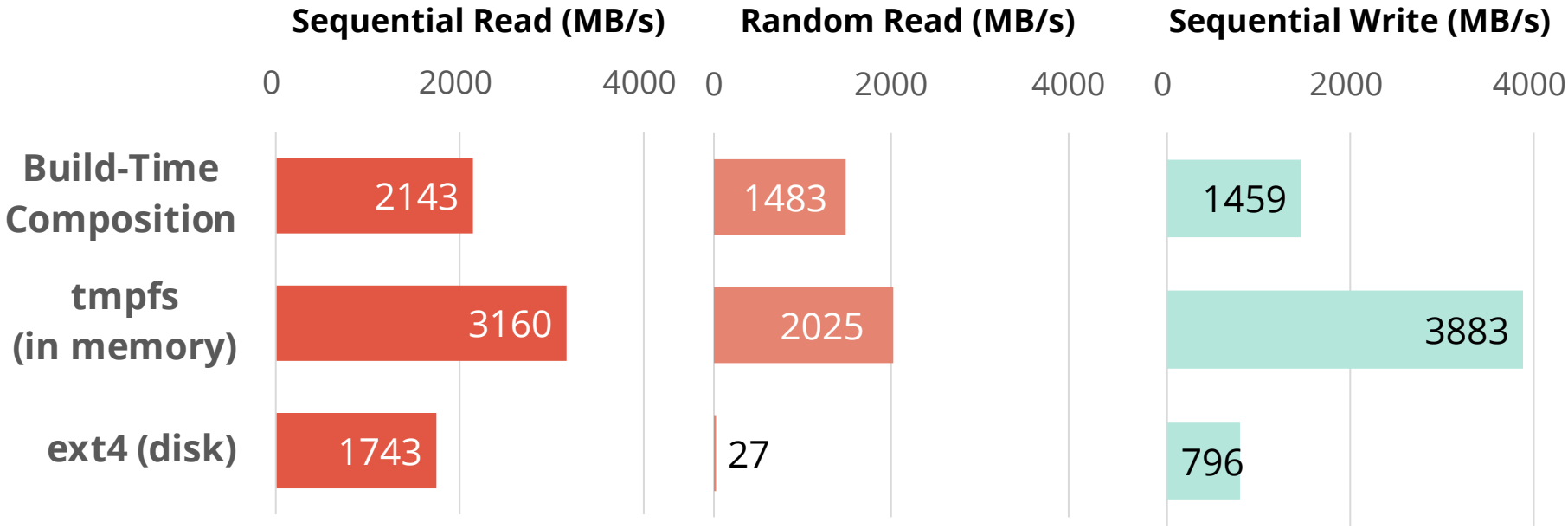
```
~/workspace/public/monaka-fs git:(main)  (0.234s)
wasmtime run -S inherit-network=y /tmp/rpc-writer.wasm "messages_by_writer.txt" "This is a message from Writer"
Wrote 29 bytes to messages_by_writer.txt

~/workspace/public/monaka-fs git:(main)  (2.694s)
awslocal s3 ls s3://test-vfs-bucket/ --recursive
2026-04-13 23:47:57         29 vfs/files/messages_by_writer.txt

~/workspace/public/monaka-fs git:(main)  (0.481s)
awslocal s3 cp "s3://test-vfs-bucket/vfs/files/messages_by_writer.txt" -
This is a message from Writer%

~/workspace/public/monaka-fs git:(main)  (0.213s)
wasmtime run -S inherit-network=y /tmp/rpc-reader.wasm "messages_by_writer.txt"
This is a message from Writer%

~/workspace/public/monaka-fs  main  ± 0
```

Performance Evaluation

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
Embedded Linux Conference

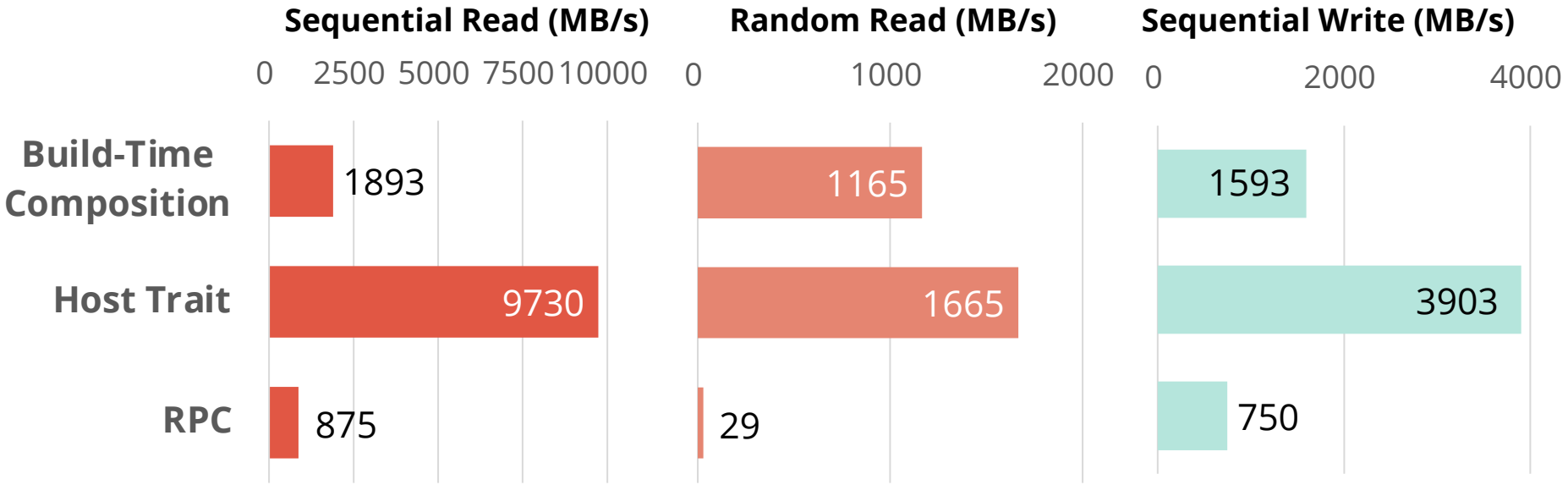
Eval 1: Build-Time Composition vs Host Filesystems

(Throughput for 100MB file size)



IO Performance: tmpfs > Build-Time Composition (proposed) > ext4
Build Time Composition achieved over 1GB/s throughput

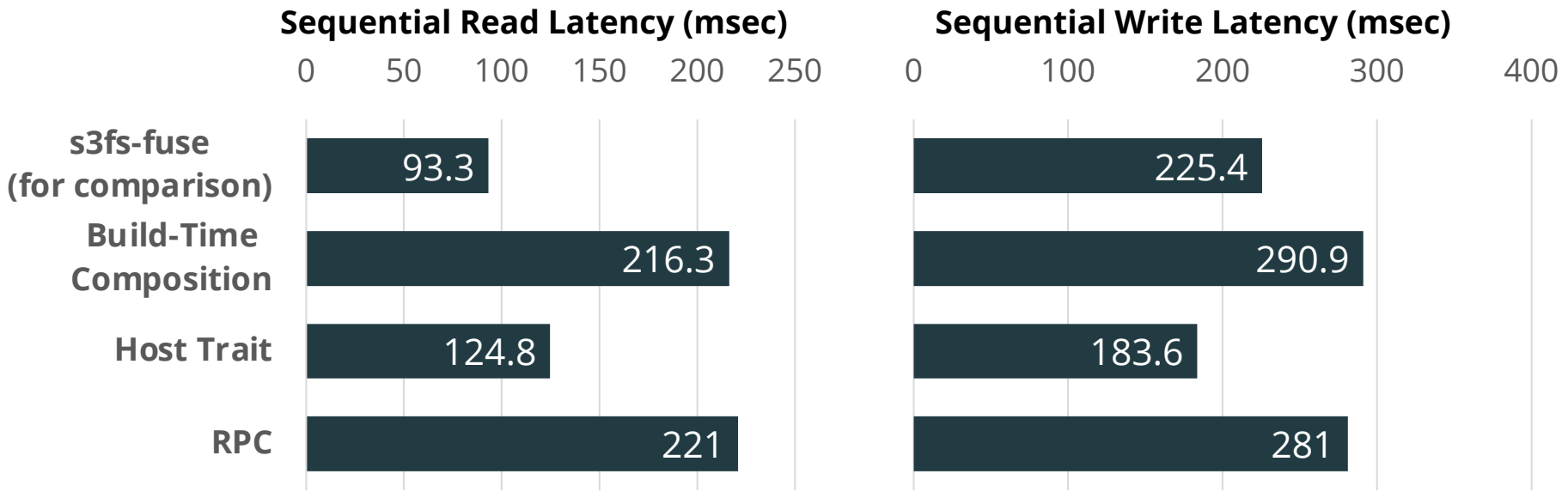
Eval 2: Comparing the Three Approaches
(Throughput for 100MB file size)



Host Trait Implementation achieved the best performance
The performance of RPC was limited especially in random read

Eval 3: S3 Sync Performance

(Latency for 1MB File Size)



IO performance: Host Trait ≒ s3fs-fuse > Build-Time > RPC

Which Approach to Choose?

Comparison & Selection Guide

	Build-Time	Host-Trait	RPC
I/O Performance	Good	**Best**	Slow
Portability	**Best** (single .wasm binary)	Good (must distribute native binary)	Good (Server should be started up beforehand)
Data Sharing	None (without S3, no sharing)	Same Process	**Cross Process**
S3 Sync	Yes	Yes	Yes
Use Case Examples	Config bundling / temporary files	Multi-app in the same process	Dynamic workers whose lifetime differ (like CI/CD)

Users can select the best method based on their use case

Limitations & Future Work (1)

- Memory constraints:
  - In-memory FS bounded by host RAM
  - Build-Time Composition & RPC: 4GiB linear memory limit (memory64 will help)

- RPC performance:
  - Caching / prefetching to reduce round-trips

Limitations & Future Work (2)

- Distributed locking for S3
  - Current: single-writer consistent
  - Multi-writer needs a distributed lock manager

- Large file streaming
  - Direct S3 streaming without loading into memory

Summary

- We built a **host-independent virtual filesystem for WebAssembly**
  - Improves security for cloud providers
  - Provides portable file operations for app developers
- Three approaches + S3 persistence:
  - Build-time composition: portable, self-contained
  - Host-trait implementation: best performance, in-process sharing
  - RPC dynamic attachment: cross-process, dynamic scaling
- Open source: https://github.com/ternbusty/monaka-fs
- Thank you! Questions?

Acknowledgement

Thank you for your support!

Japan Advanced Institute of Science and Technology

Dr. Shinoda & Dr. Uda Lab Members

Thank you!

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Embedded Linux Conference