Continuation-Centric Computing with Arca

Akshay Srivatsan Yuhan Deng Katherine Mohr Emma Sudo
Sebastian Ingino Francis Chua Keith Winstein

Stanford University

Outline

Continuation-Centric Computing with Arca 1
What is a continuation? 2
Why use continuations? 5
How do programmers use continuations? 9
Arca: a Continuation-Centric OS 18
Bibliography 31

What is a continuation?

Continuations in the abstract

- “the rest of the function”

Continuations in the abstract

- “the rest of the function”

Original function:

$$f(x) = g(h(x))$$

Continuations in the abstract

- “the rest of the function”

Original function:

$$f(x) = g(h(x))$$

- to do $f(x)$:
  - do $h(x)$
  - then do g on the result

Continuations in the abstract

- “the rest of the function”

Original function:

$$f(x) = g(h(x))$$

- to do $f(x)$:
  - do $h(x)$
  - then do g on the result

Defined with a continuation:

$$f'(x) = h'(x, f_{\text{cont}})$$

$$\text{where } f_{\text{cont}}(y) = g'(y)$$

Continuations in the abstract

- “the rest of the function”

Original function:

$$f(x) = g(h(x))$$

- to do $f(x)$:
  - do $h(x)$
  - then do g on the result

Defined with a continuation:

$$f'(x) = h'(x, f_{\text{cont}})$$

$$\text{where } f_{\text{cont}}(y) = g'(y)$$

- to do $f'(x)$:
  - do $h'(x)$ and then f_{cont}
- to do f_{cont}, do g'

Continuations in code

```
def do_things(x):

  y = thing_1(x)

  return thing_2(y)
```

Continuations in code

```
def do_things(x):

  # first: do thing_1
  y = thing_1(x)

  # second: do thing_2
  return thing_2(y)
```

Continuations in code

```
def do_things(x):

  # first: do thing_1
  y = thing_1(x)

  # second: do thing_2
  return thing_2(y)
```

```
def do_things(x):


  def cont(y):
    return thing_2(y)


  return thing_1(x, cont)
```

Continuations in code

```
def do_things(x):

  # first: do thing_1
  y = thing_1(x)

  # second: do thing_2
  return thing_2(y)
```

```
def do_things(x):

  # eventually: do thing 2
  def cont(y):
    return thing_2(y)

  # but for now: do thing 1
  return thing_1(x, cont)
```

Why use continuations?

Examples in Serverless Computing

- Fix[1] (EuroSys '26): I/O-compute separation
- Quicksand[2] (NSDI 25): "logical processes" → "proclets"
- Dandelion[3] (SOSP '25): computation-communication separation

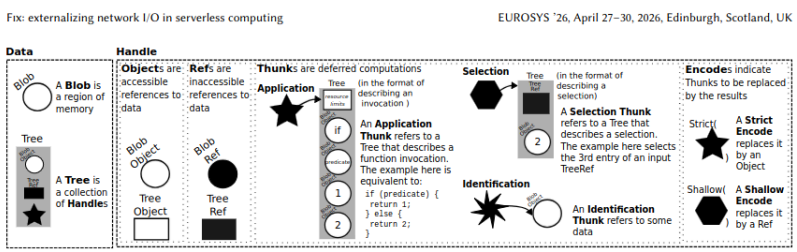


Figure 1. Fix represents function invocations and data dependencies with a unified serializable format.

```
void preprocessing_stage(qs::ShardedVector<Img>& inputs,
                         qs::ShardedQueue<Img>& outputs) {
 auto f = [outputs](Img i) { outputs.push(preprocess(i)); };
 auto scaling_policy = qs::batch::MinimizeQueue(outputs);
 qs::batch::ForAll(inputs, f, scaling_policy);
}
```

Figure 3: An example of an ML training pipeline illustrating the decomposition of the application into resource proclets using Quicksand's high-level abstractions. It highlights the preprocessing stage, where distributed workers process input images read (with prefetching) from a sharded vector and output to a sharded queue. Quicksand's batch computing framework dynamically scales the number of workers based on the output queue size to match the training stage's consumption rate.


```
composition RenderLogs(AccessToken) => HTMLOutput {
    Access(AccessToken = all AccessToken)
        => (AuthRequest = HTTPRequest);
    HTTP(Request = each AuthRequest)
        => (AuthResponse = Response);
    FanOut(HTTPResponse = all AuthResponse)
        => (LogRequests = HTTPRequests);
    HTTP(Request = each LogRequests)
        => (LogResponses = Response);
    Render(HTTPResponses = all LogResponses)
        => (HTMLOutput = HTMLOutput);
}
```

Listing 2. Log processing application from Listing 1 expressed as a Dandelion composition DAG using our DSL.

[1] Deng et al., "Fix: Externalizing Network I/O in Serverless Computing".

[2] Ruan et al., "Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing".

[3] Kuchler et al., "Unlocking True Elasticity for the Cloud-Native Era with Dandelion".

Continuations convert from the “logical” order of operations to the “physical” order of operations.

- logical = good for developers
- physical = good for schedulers

Schedulers benefit from **different abstractions** than developers.

Continuation ≈ paused task

Examples of (implicit) continuations:

- VM snapshots
- CRIU checkpoints
- Linux processes waiting on disk or network I/O
- idle coroutines

How do programmers use continuations?

Three methods of implementing continuations:

1. Explicit State Saving
2. Closures + Callbacks
3. Language Support

Method 1: explicit state saving

- programmer writes code to serialize/deserialize program state
- common in C and WebAssembly
- used in Fix (2026)

Method 1: explicit state saving

- programmer writes code to serialize/deserialize program state
- common in C and WebAssembly
- used in Fix (2026)

Systemd

```
// Version 261.1, `src/core/main.c`:
OPTION_LONG("deserialize", "FD", /* help= */ NULL):
```

Method 1: explicit state saving

- programmer writes code to serialize/deserialize program state
- common in C and WebAssembly
- used in Fix (2026)

Systemd

```
// Version 261.1, `src/core/main.c`:
OPTION_LONG("deserialize", "FD", /* help= */ NULL):
```

OpenSSH

```
// Version 10.3p1, `sshd.c`:
rexec_argv = xcalloc(rexec_argc + 3, sizeof(char *));
rexec_argv[0] = options.sshd_session_path;
for (i = 1; i < (u_int)rexec_argc; i++) {
  rexec_argv[i] = saved_argv[i];
}
rexec_argv[rexec_argc++] = "-R";
rexec_argv[rexec_argc] = NULL;
```

Method 1: explicit state saving

- programmer writes code to serialize/deserialize program state
- common in C and WebAssembly
- used in Fix (2026)

Systemd

```
// Version 261.1, `src/core/main.c`:
OPTION_LONG("deserialize", "FD", /* help= */ NULL):
```

OpenSSH

```
// Version 10.3p1, `sshd.c`:
rexec_argv = xcalloc(rexec_argc + 3, sizeof(char *));
rexec_argv[0] = options.sshd_session_path;
for (i = 1; i < (u_int)rexec_argc; i++) {
  rexec_argv[i] = saved_argv[i];
}
rexec_argv[rexec_argc++] = "-R";
rexec_argv[rexec_argc] = NULL;
```

Nginx

```
// Version 1.31.2, `http/ngx_http_request.c`:
rev->handler = ngx_http_wait_request_handler;
c->write->handler = ngx_http_empty_handler;
```

Method 1: explicit state saving

- programmer writes code to serialize/deserialize program state
- common in C and WebAssembly
- used in Fix (2026)
- **high developer burden**

Method 2: closures + callbacks

- continuations are explicit callback functions
- uses language-level closures for state capture
- common in JavaScript and C++
- used in Quicksand (2025)

Method 2: closures + callbacks

- continuations are explicit callback functions
- uses language-level closures for state capture
- common in JavaScript and C++
- used in Quicksand (2025)

Node.js

```
// Version 26.5.0, example code for `net.createConnection`:
const client = net.createConnection({ port: 8124 }, () => {
  client.write('world!\r\n');
});
```

Method 2: closures + callbacks

- continuations are explicit callback functions
- uses language-level closures for state capture
- common in JavaScript and C++
- used in Quicksand (2025)
- closures are **not generally serializable**
  - must be resumed within the same language runtime
  - extra work needed to migrate across machines
- developer overhead: **“callback hell”**

Method 3: language support

- continuations are implicitly captured by the language runtime
- common in Rust and Go (and modern JavaScript/C++)
- used in Dandelion (2025)

Method 3: language support

- continuations are implicitly captured by the language runtime
- common in Rust and Go (and modern JavaScript/C++)
- used in Dandelion (2025)

Rust (async/await)

```rust
// Tokio mini-redis tutorial:
#[tokio::main]
async fn main() -> Result<()> {
  let mut client = client::connect("127.0.0.1:6379").await?;
  client.set("hello", "world".into()).await?;
  let result = client.get("hello").await?;
  println!("got value from the server; result={:?}", result);
  Ok(())
}
```

Method 3: language support

- continuations are implicitly captured by the language runtime
- common in Rust and Go (and modern JavaScript/C++)
- used in Dandelion (2025)

Rust (async/await)

```rust
// Tokio mini-redis tutorial:
#[tokio::main]
async fn main() -> Result<()> {
  let mut client = client::connect("127.0.0.1:6379").await?;
  client.set("hello", "world".into()).await?;
  let result = client.get("hello").await?;
  println!("got value from the server; result={:?}", result);
  Ok(())
}
```

Go (coroutines)

```go
// https://go.dev/tour/concurrency/1
func say(s string) {
  for i := 0; i < 5; i++ {
    time.Sleep(100 * time.Millisecond)
    fmt.Println(s)
  }
}
```

```go
func main() {
  go say("world")
  say("hello")
}
```

Method 3: language support

- continuations are implicitly captured by the language runtime
- common in Rust and Go (and modern JavaScript/C++)
- used in Dandelion (2025)
- coroutines are still **not serializable**
  - tied to a particular instance of a language runtime

Existing Methods

1. Explicit State Saving (high developer overhead)
2. Closures + Callbacks (“callback hell”)
3. Language Support (not serializable/migratable)

Existing Methods

1. Explicit State Saving (high developer overhead)
2. Closures + Callbacks (“callback hell”)
3. Language Support (not serializable/migratable)

Competing Goals

Developers want compatibility: code that looks “normal”

- continuation capture needs to be **transparent to developers**

Existing Methods

1. Explicit State Saving (high developer overhead)
2. Closures + Callbacks (“callback hell”)
3. Language Support (not serializable/migratable)

Competing Goals

Developers want compatibility: code that looks “normal”

- continuation capture needs to be **transparent to developers**

Schedulers want granularity: flexibility to schedule freely

- continuations need to be **efficient to capture**

Existing methods make different trade-offs.

An OS kernel can provide efficient, transparent continuation capture.

Arca: a Continuation-Centric OS

Arca's Approach

1. Reduce processes to the bare minimum.

- registers
- address space
- file descriptors[1]

[1] Actually "value descriptors"; see paper for details.

Arca's Approach

1. Reduce processes to the bare minimum.

- registers
- address space
- file descriptors[1]

2. Reify paused processes as userspace-visible continuations.

- **Efficiency:** captured "for free" on every system call
- **Transparency:** looks to userspace like a blocking system call

[1] Actually "value descriptors"; see paper for details.

Efficient continuation capture

- Goal: **continuations are fast to capture and resume**

Efficient continuation capture

- Goal: **continuations are fast to capture and resume**
  - no machine-dependent state (e.g., inodes)
  - no shared mutable state (e.g., shared-memory threads)
  - no unstructured I/O or IPC (e.g., kernel bypass)

Efficient continuation capture

- Goal: **continuations are fast to capture and resume**
  - no machine-dependent state (e.g., inodes)
  - no shared mutable state (e.g., shared-memory threads)
  - no unstructured I/O or IPC (e.g., kernel bypass)
- Solution: **process creation/deletion does not need synchronization**
  - Processes look more like pure functions.
  - Global state handled through “effect handlers”: see paper.
  - No locking, TLB shootdowns, etc.

Transparent continuation capture

- Goal: **continuations are compatible with existing code**

Transparent continuation capture

- Goal: **continuations are compatible with existing code**
  - Developers prefer straight-line code.
  - Developers prefer using languages they already know.

Transparent continuation capture

- Goal: **continuations are compatible with existing code**
  - Developers prefer straight-line code.
  - Developers prefer using languages they already know.

```
def foo(x):
  y = bar(x)
  z = baz(y)
  return qux(z)
```

```
def foo(x, finish):
  def foo_cont1(y):
    def foo_cont2(z):
      qux(z, finish)
    baz(y, foo_cont2)
  bar(x, foo_cont1)
```

```
foo :: a -> IO a
foo x = do
  y <- bar x
  z <- baz y
  qux z
```

Transparent continuation capture

- Goal: **continuations are compatible with existing code**
  - Developers prefer straight-line code.
  - Developers prefer using languages they already know.

```
def foo(x):
  y = bar(x)
  z = baz(y)
  return qux(z)
```

```
def foo(x, finish):
  def foo_cont1(y):
    def foo_cont2(z):
      qux(z, finish)
    baz(y, foo_cont2)
  bar(x, foo_cont1)
```

```
foo :: a -> IO a
foo x = do
  y <- bar x
  z <- baz y
  qux z
```

- Solution: **continuation capture via system call**

Arca's system call interface

New system calls:

`get_cc()` Returns the rest of the current program as a new function.

Arca's system call interface

New system calls:

`get_cc()` Returns the rest of the current program as a new function.

`call_cc(f)` Calls another function, passing the rest of the current function as a callback.

Call with Current Continuation: Original Program

```
def thumbnail_image(filename):
  input = get(filename)
  output = thumbnail(input)
  put(filename, output)
```

```
def get(name):
  sys_read_file(name)

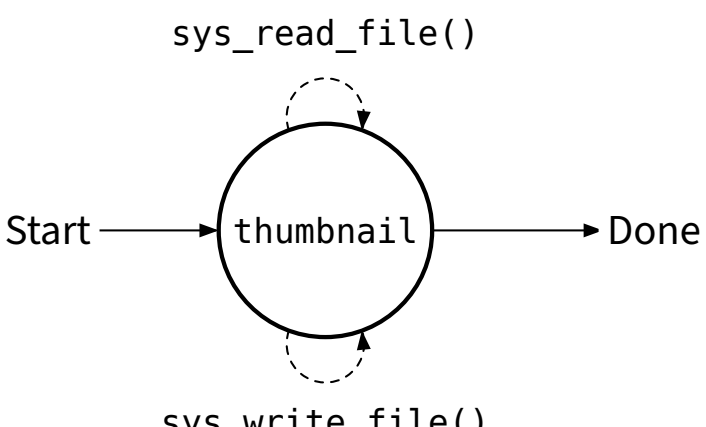
def put(name, data):
  sys_write_file(name, data)
```

Call with Current Continuation: Original Program

```
def thumbnail_image(filename):
  input = get(filename)
  output = thumbnail(input)
  put(filename, output)
```

```
def get(name):
  sys_read_file(name)

def put(name, data):
  sys_write_file(name, data)
```



Call with Current Continuation: Continuation-Centric

```
# this function is unchanged
def thumbnail_image(filename):
  input = get(filename)
  output = thumbnail(input)
  put(filename, output)
```

```
# "effects" are handled by provider
def get(name):
  sys_call_cc("read", name)

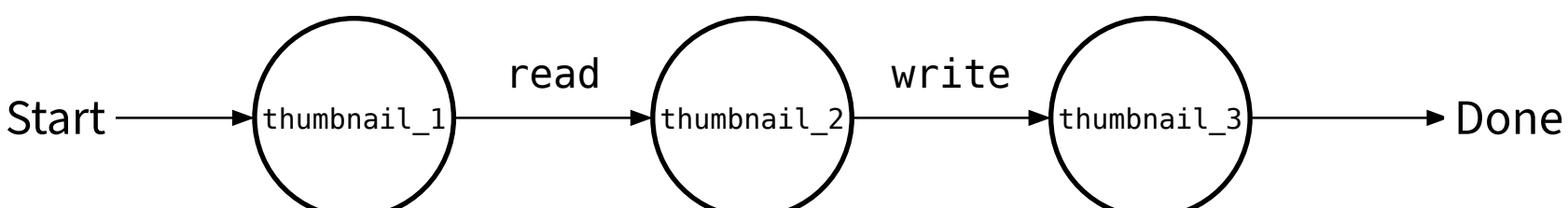
def put(name, data):
  sys_call_cc("write", name, data)
```

Call with Current Continuation: Continuation-Centric

```
# this function is unchanged
def thumbnail_image(filename):
  input = get(filename)
  output = thumbnail(input)
  put(filename, output)
```

```
# "effects" are handled by provider
def get(name):
  sys_call_cc("read", name)

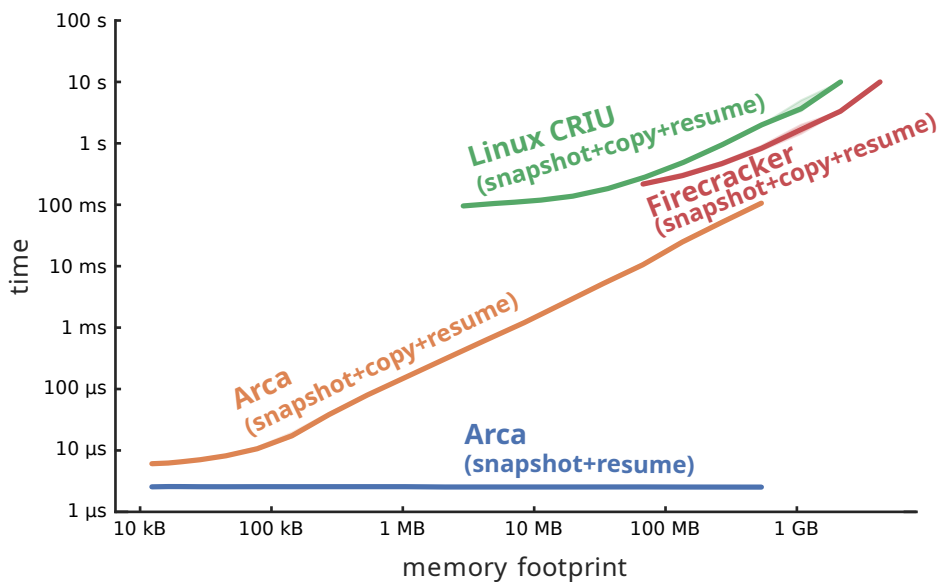
def put(name, data):
  sys_call_cc("write", name, data)
```



Zero-copy continuations are “free.”

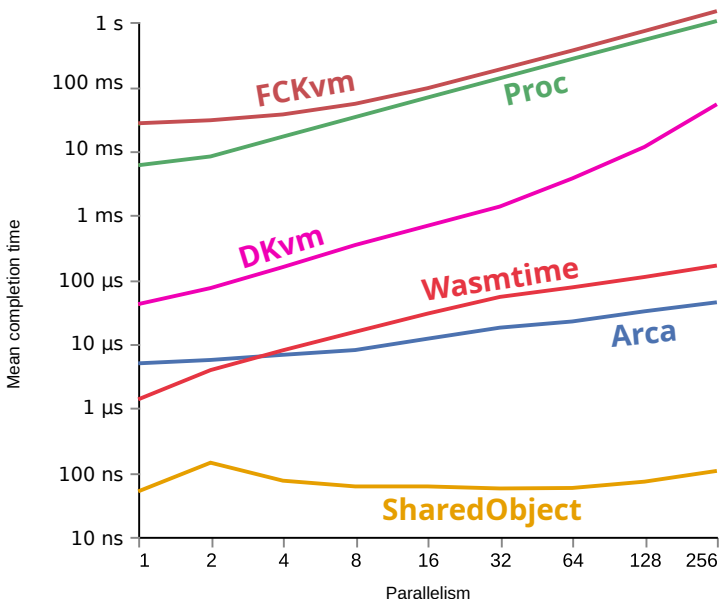
- Most continuations are only resumed once.

- Copy $\propto$ bandwidth
- Zero-copy $\approx 2\mu s$
- $O(n) \rightarrow O(1)$



Continuation-centric processes are fast to create.

- Minimal processes are comparable in performance to SFI.
- startup time $\propto$ synchronization
  - allocation, TLB shootdowns
- continuation-centric design avoids synchronization



Serializable continuations enable new optimizations.

Workload: image thumbnailing server

Serializable continuations enable new optimizations.

Workload: image thumbnailing server

Traditional (total: 106 ms)



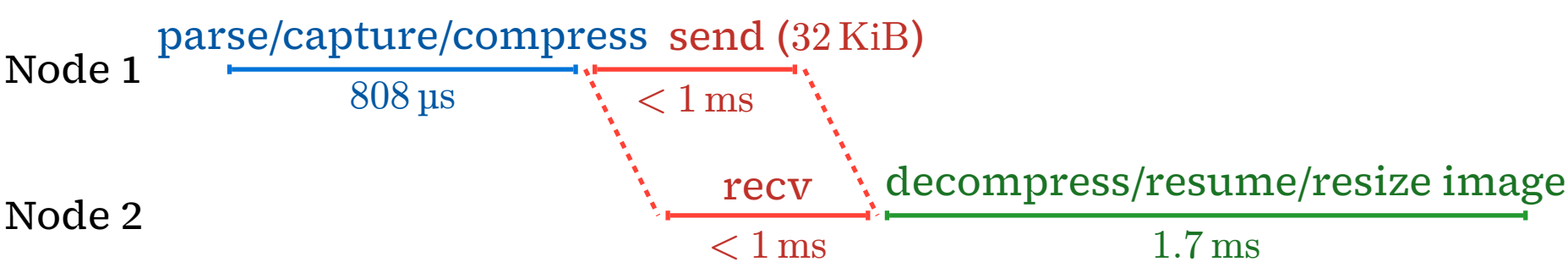
Serializable continuations enable new optimizations.

Workload: image thumbnailing server

Traditional (total: 106 ms)



Continuation-Centric (total: 2.82 ms)



Limitations

Designed for FaaS workloads short-lived functions with little state

Fork/Join parallelism good for MapReduce-style workloads

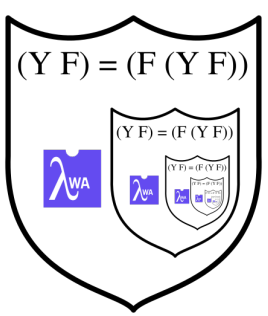
Not Linux-compatible cannot run unmodified Linux software

- but *can* run unmodified WebAssembly+WASI applications, e.g., `ffmpeg.wasm`, via `wasm2c`
- can run some recompiled POSIX applications via `musl`

Conclusion

An OS kernel can provide efficient, transparent continuation capture.

Arca is a proof-of-concept continuation-centric kernel.



https://github.com/fix-project

srivatsan@cs.stanford.edu

Acknowledgements

Our thanks to:

- our shepherd, Malte Schwarzkopf, and our reviewers
- our collaborators at Stanford:
  - Dawson Engler
  - Haibib Kerim
  - Matthew Sotoudeh
  - Michael Paper
  - Thea Rossman
  - Zachary Yedidia
- Ariel Szekely, Frans Kaashoek, and the rest of PDOS at MIT
- Ana Klimovic and Timothy Roscoe from the ETH Systems Group (ETH Zurich)

Bibliography

[1] Y. Deng *et al.*, “Fix: externalizing network I/O in serverless computing,” in *Proceedings of the 21st European Conference on Computer Systems*, in EuroSys '26. McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK: Association for Computing Machinery, 2026, pp. 2260–2275. doi: 10.1145/3767295.3769387.

[2] Z. Ruan *et al.*, “Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing,” in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, Philadelphia, PA: USENIX Association, Apr. 2025, pp. 147–165. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/ruan

[3] T. Kuchler *et al.*, “Unlocking True Elasticity for the Cloud-Native Era with Dandelion,” in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, in SOSP '25. Lotte Hotel World, Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 944–961. doi: 10.1145/3731569.3764803.

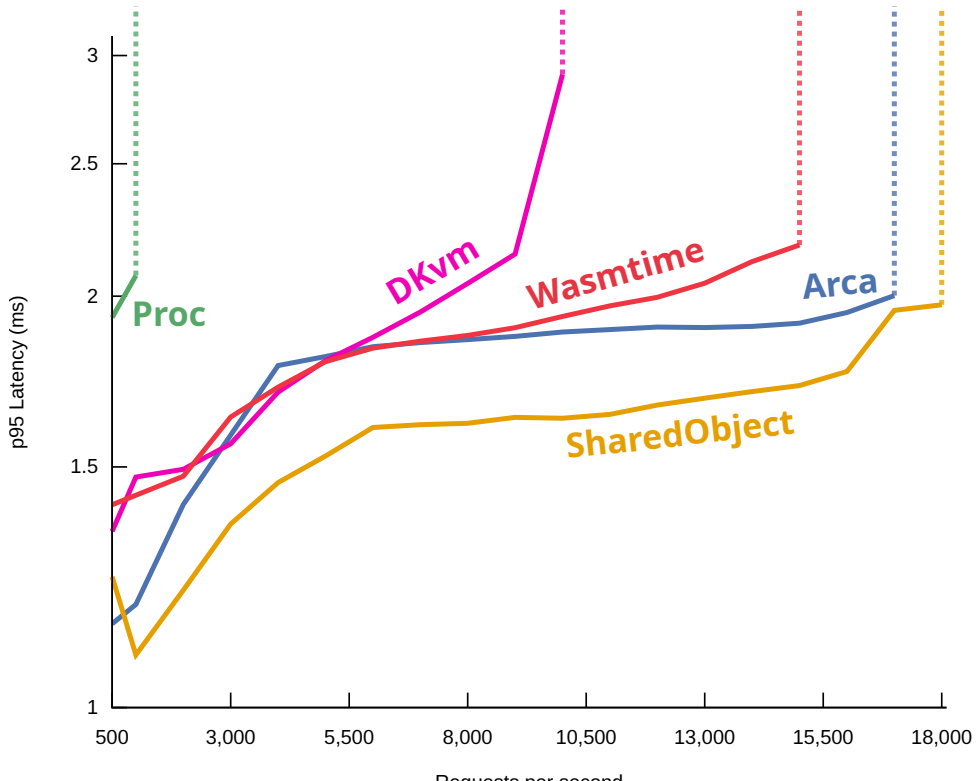
Isolation Primitives

Isolation primitive	create/destroy (μs)	snapshot/resume (μs)
MicroVM	✗ 742000	✗ 217000
Linux process	✗ 540000	✗ 283000
WebAssembly	✓ 110.	✗ n/a
Arca (our work)	✓ 32.2	✓ 2.55

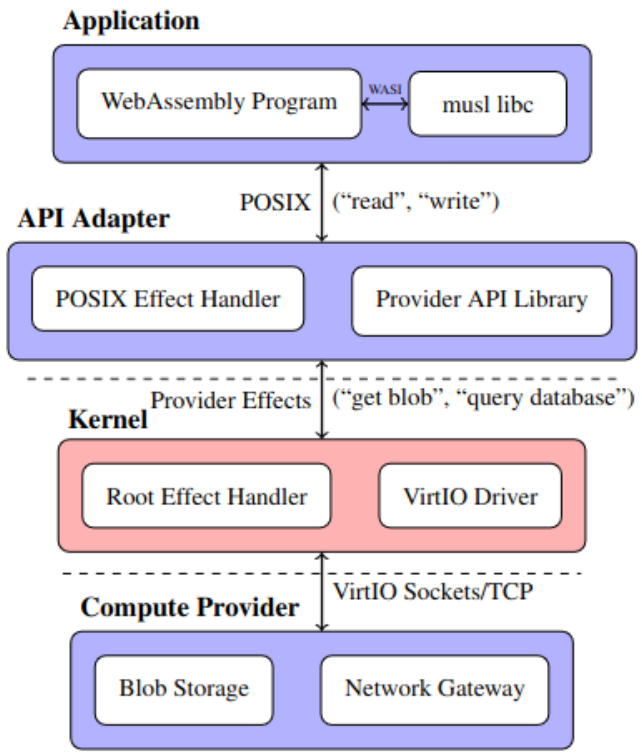
Continuation overhead on I/O heavy work

Mode	System	p50 Latency (ms)	p99 Latency (ms)
Workers	Arca	1.642	2.217
	Apache	1.737	2.067
CGI-Style	Arca	3.665	5.390
	Apache	16.53	19.36

Open-Loop Request Handling



Compatibility with WebAssembly/WASI



Supported POSIX APIs

POSIX Function	Status	Notes
`open`	✓	Translated to provider-native storage effects.
`close`	✓	
`read`	✓	
`write`	✓	
`lseek`	✓	
`socket`	○	Provider must provide low-level access to networking,
`connect`	○	
`send`	○	
`recv`	○	
`listen`	○	

POSIX Function	Status	Notes
`accept`	○	which makes continuation capture difficult.
`shm_open`	✗	No shared mutable memory support.
`mmap` `MAP_SHARED`	✗	

System Call API

System Call	Description
`create_blob`	Construct a byte array.
`create_tuple`	Construct a collection.
`create_page`	Allocate a page of memory.
`create_table`	Allocate a page table.
`create_fn`	Define a new funclet.
`read`	Read a value into memory.
`write`	Update a value in kernel memory.
`clone`	Make a CoW clone of a value.

System Call	Description
`drop`	Delete a value.
`mmap`	Modify the address space.
`get_arg`	Get the next argument.
`exit`	Return a value.
`call`	Call another function.
`get_cc`	Capture the continuation of this function.
`call_cc`	Tail-call another function, passing this function's continuation as an argument.