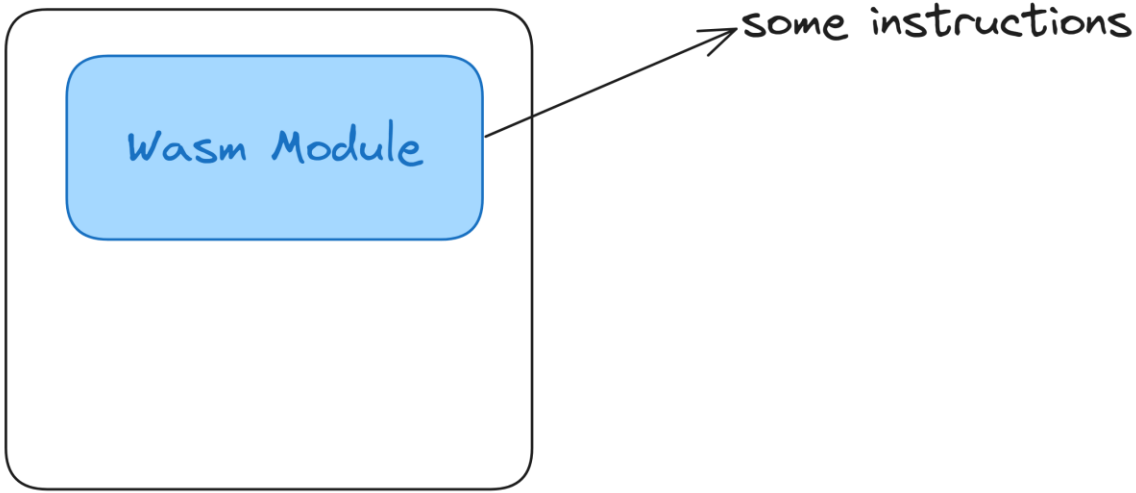
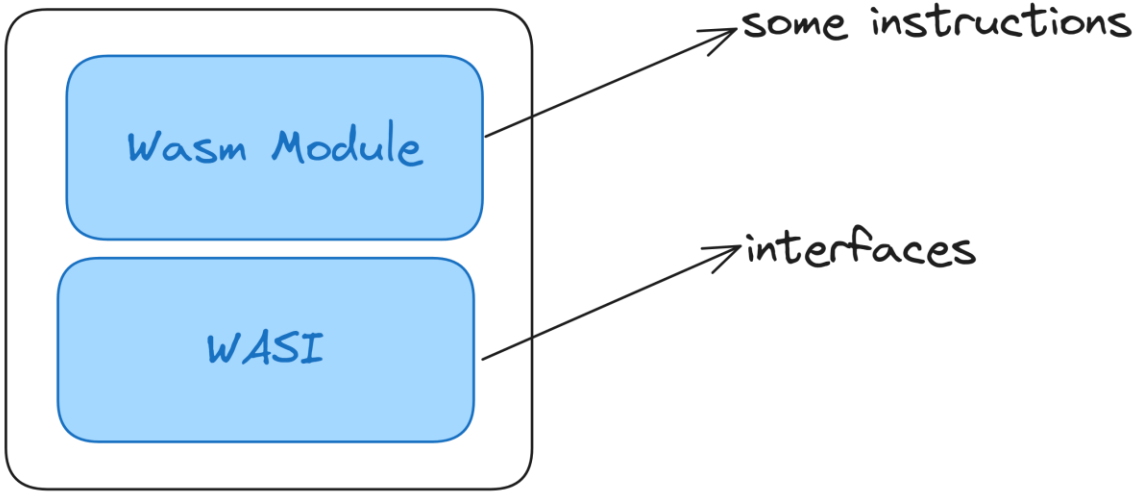
Components in Wasm

Components in Wasm



Components in Wasm

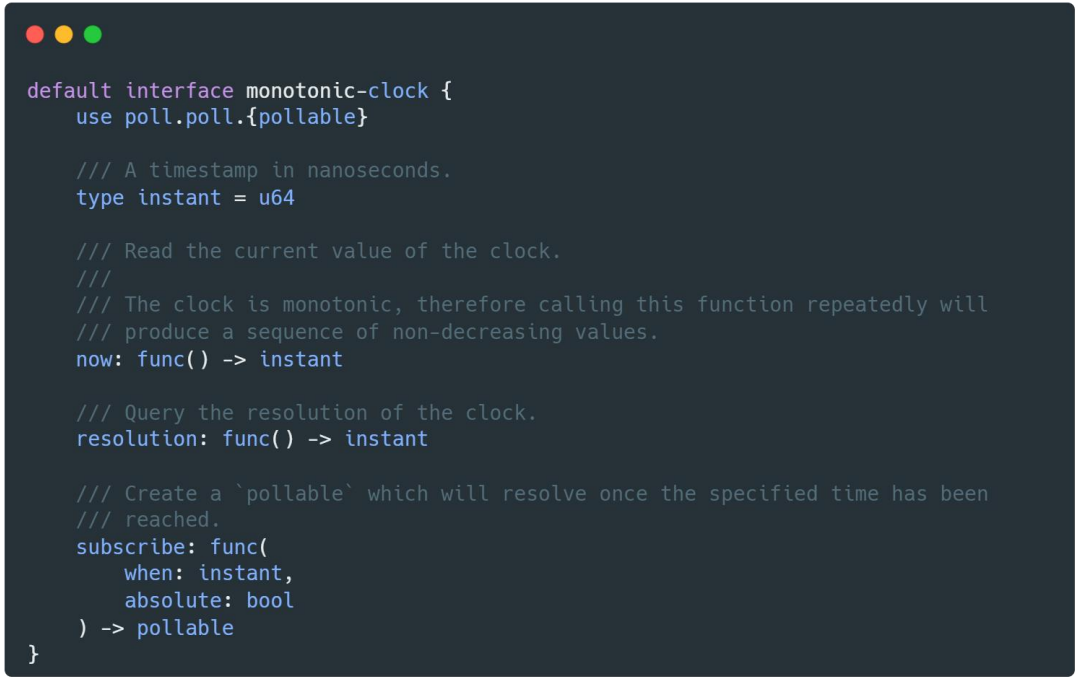


Components in Wasm

- We don’t want to talk about the what goes in and comes out of a function instead about types, functions, methods, and namespaces
- How does component model handle this?

  Wasm Interface Types

Components in Wasm



```
default interface monotonic-clock {
    use poll.poll.{pollable}

    /// A timestamp in nanoseconds.
    type instant = u64

    /// Read the current value of the clock.
    ///
    /// The clock is monotonic, therefore calling this function repeatedly will
    /// produce a sequence of non-decreasing values.
    now: func() -> instant

    /// Query the resolution of the clock.
    resolution: func() -> instant

    /// Create a `pollable` which will resolve once the specified time has been
    /// reached.
    subscribe: func(
        when: instant,
        absolute: bool
    ) -> pollable
}
```

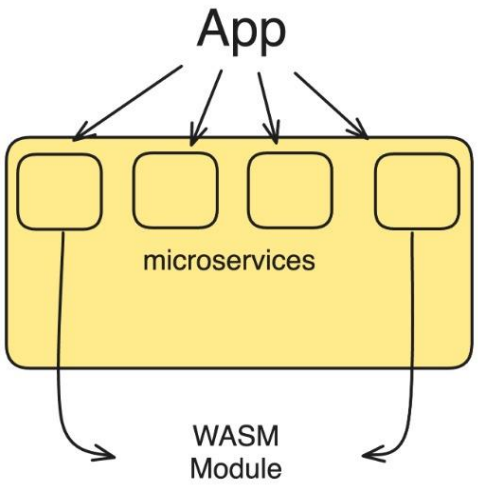
There are also Worlds

- Larger ecosystem of worlds
-
```
default world some-world {
  import : self.
  export : func() }
```

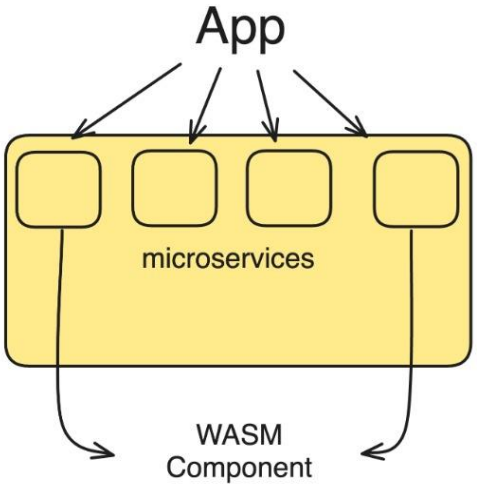
Component Model

- separately-compiled components built from Wasm modules
- component instances fully encapsulate their linear memories, tables, globals
- destructors for component instances are deterministic

But the Wasm Module?



Interaction
Communication with
other modules
Iterative changes
Host machine contract



But the Wasm Module?

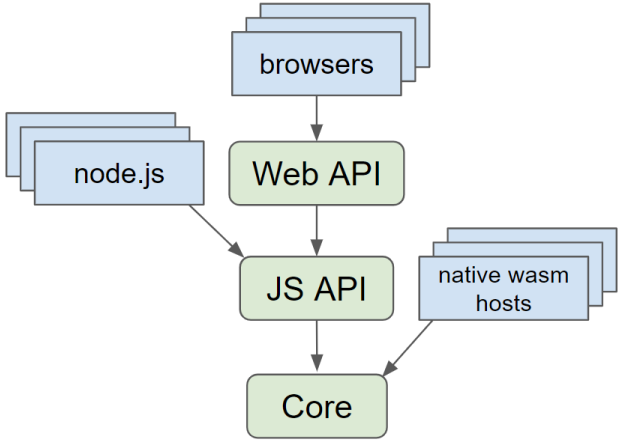
- separate compilation and deployment
- fully explicit dependencies
- black-box reuse
- external composition by independent parties

Module Linking

```
(module $COMPOUND
  (module $LIBC … )
  (module $APP_A
    (import “libc” (module $LIBC …))
    (instance $libc (instantiate $LIBC))
    …
  )
  (module $APP_B
    (import “libc” (module $LIBC …))
    (instance $libc (instantiate $LIBC))
    …
  )
  (instance $app_a (instantiate $APP_A (import “libc” (module $LIBC))))
  (instance $app_b (instantiate $APP_B (import “libc” (module $LIBC))))
)
```

[GC, 2021]

Module Linking

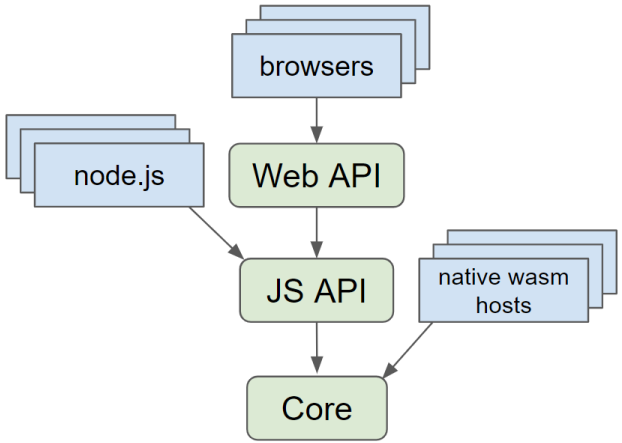


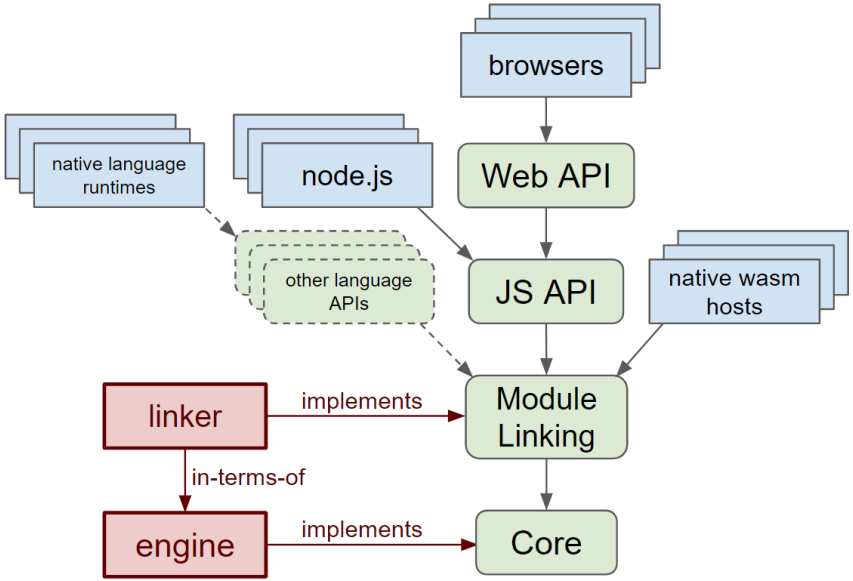
[GC, 2021]

Module Linking

[GC, 2021]

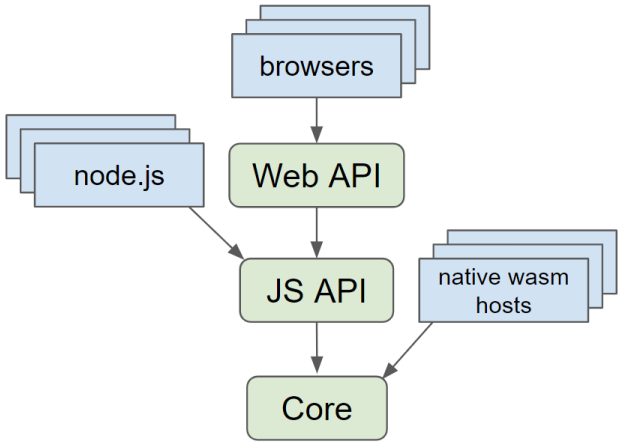
Module Linking

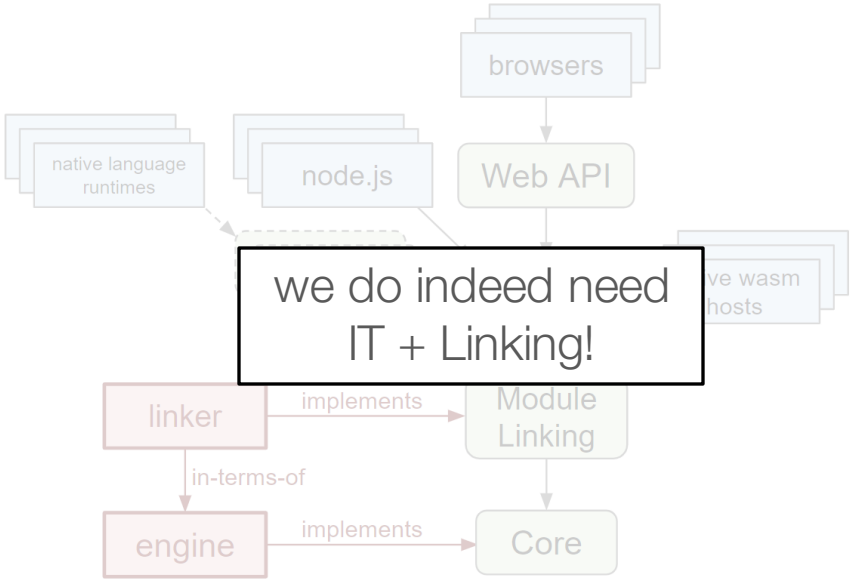




[GC, 2021]

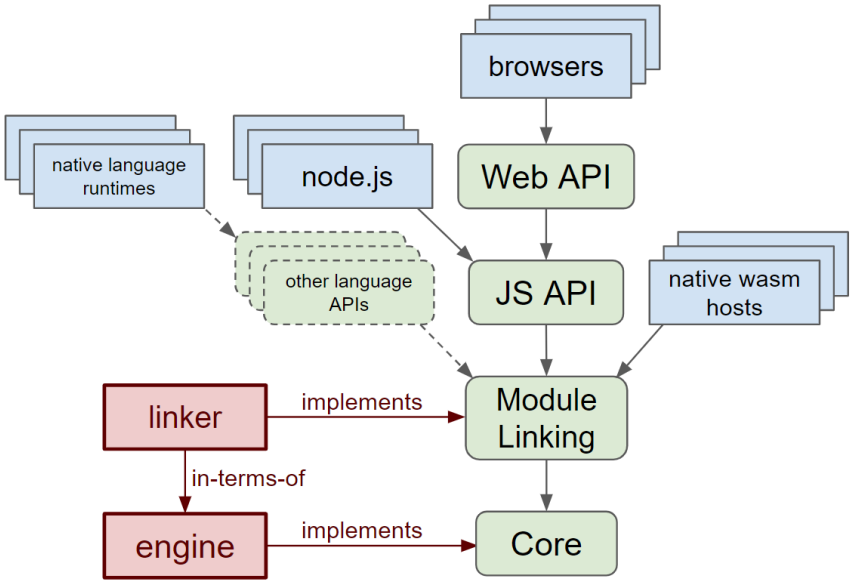
Module Linking





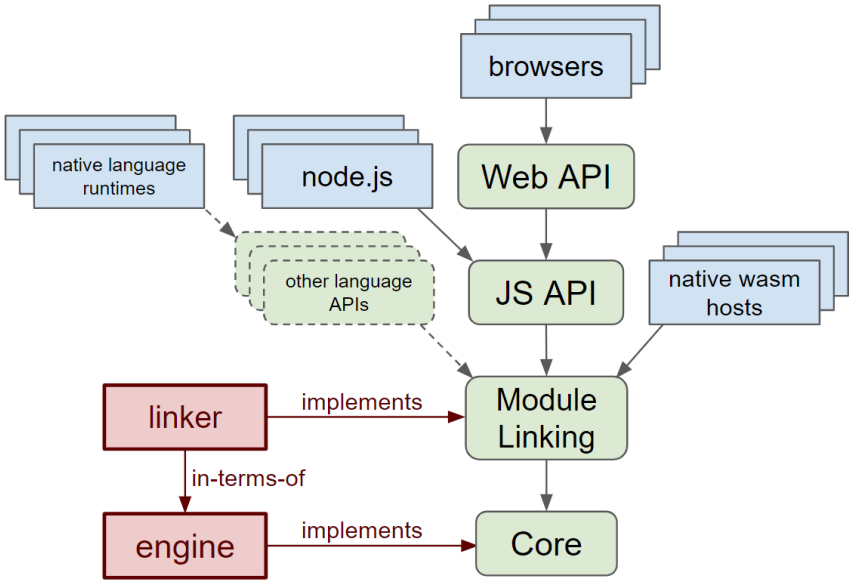
[GC, 2021]

Module Linking



[GC, 2021]

Module Linking



[GC, 2021]

Component Model

The way it can all come truly come together is

Core + (Module Linking + Interface Types)

```
(adapter module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module
    (import "libc" "malloc" (func (param i32) (result i32)))
    ...
    (func (export "run") (param i32 i32) (result i32 i32) ...)
  )
  (instance $core (instantiate $CORE (import “libc” “malloc” (func $libc “malloc”))))
  (adapter_func (export “run”) (param string) (result string)
    … lower param
    call (func $core “run”)
    … lift result
  )
)
```

[GC, 2021]

Component Model

The way it can all come truly come together is

Core + (Module Linking + Interface Types)

```
(adapter module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module
    (import "libc" "malloc" (func (param i32) (result i32)))
    ...
    (func (export "run") (param i32 i32) (result i32 i32) ...)
  )
  (instance $core (instantiate $CORE (import “libc” “malloc” (func $libc “malloc”))))
  (adapter_func (export “run”) (param string) (result string)
    … lower param
    call (func $core “run”)
    … lift result
  )
)
```

Interface Type

[GC, 2021]

Component Model

The way it can all come truly come together is

Core + (Module Linking + Interface Types)

```
(adapter module
  (import “libc” (module $LIBC …))
  (instance $libc (instantiate $LIBC))
  (module
    (import "libc" "malloc" (func (param i32) (result i32)))
    ...
    (func (export "run") (param i32 i32) (result i32 i32) ...)
  )
  (instance $core (instantiate $CORE (import “libc” “malloc” (func $libc “malloc”))))
  (adapter_func (export “run”) (param string) (result string)
    … lower param
    call (func $core “run”)
    … lift result
  )
)
```

Module Linking

[GC, 2021]

WIT

- interfaces + worlds

(We don’t go over all aspects of writing WIT or the design and encourage the motivated listener to checkout the WIT design document)

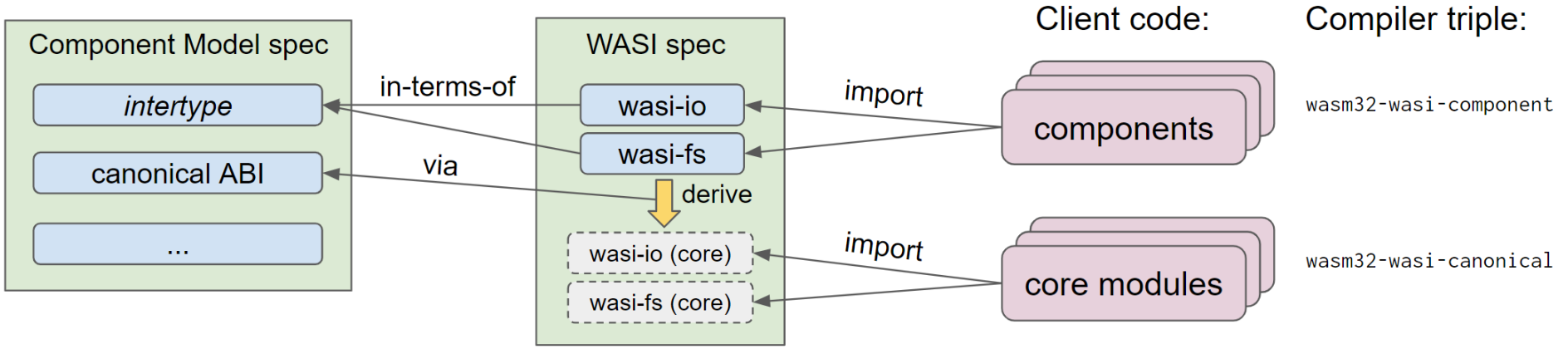
WIT

- interfaces + worlds

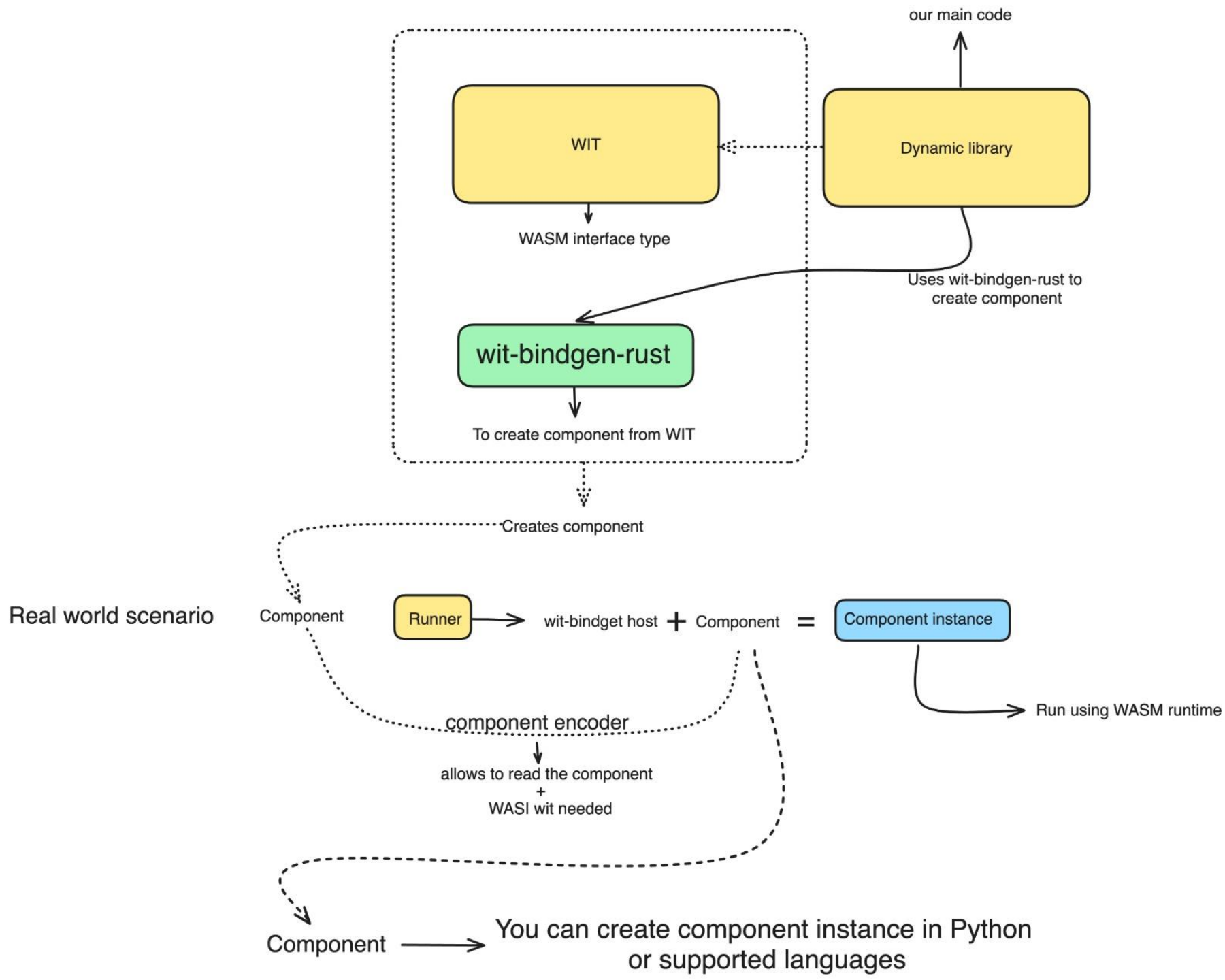
```
package local:demo;

world command {
  import wasi:filesystem/filesystem;
  import wasi:random/random;
  import wasi:clocks/monotonic-clock;
  // ...

  export main: func(args: list<string>);
}
```

(We don't go over all aspects of writing WIT or the design and encourage the motivated listener to checkout the WIT design document)

With WASI



[GC, 2021]


our main code
WIT
Dynamic library
WASM interface type
Uses wit-bindgen-rust to create component
wit-bindgen-rust
To create component from WIT
Creates component
Real world scenario
Component
Runner
wit-bindget host
+
Component
=
Component instance
Run using WASM runtime
component encoder
allows to read the component
+
WASI wit needed
Component
You can create component instance in Python or supported languages