# An Edge KV Store built with Wasm Components

# Introductions

Dan Gohman
Software Engineer
Wasm Team, Fastly

Ben Young
Software Engineer
Storage Products, Fastly

(It's the same picture)

# What is an Edge KV Store?

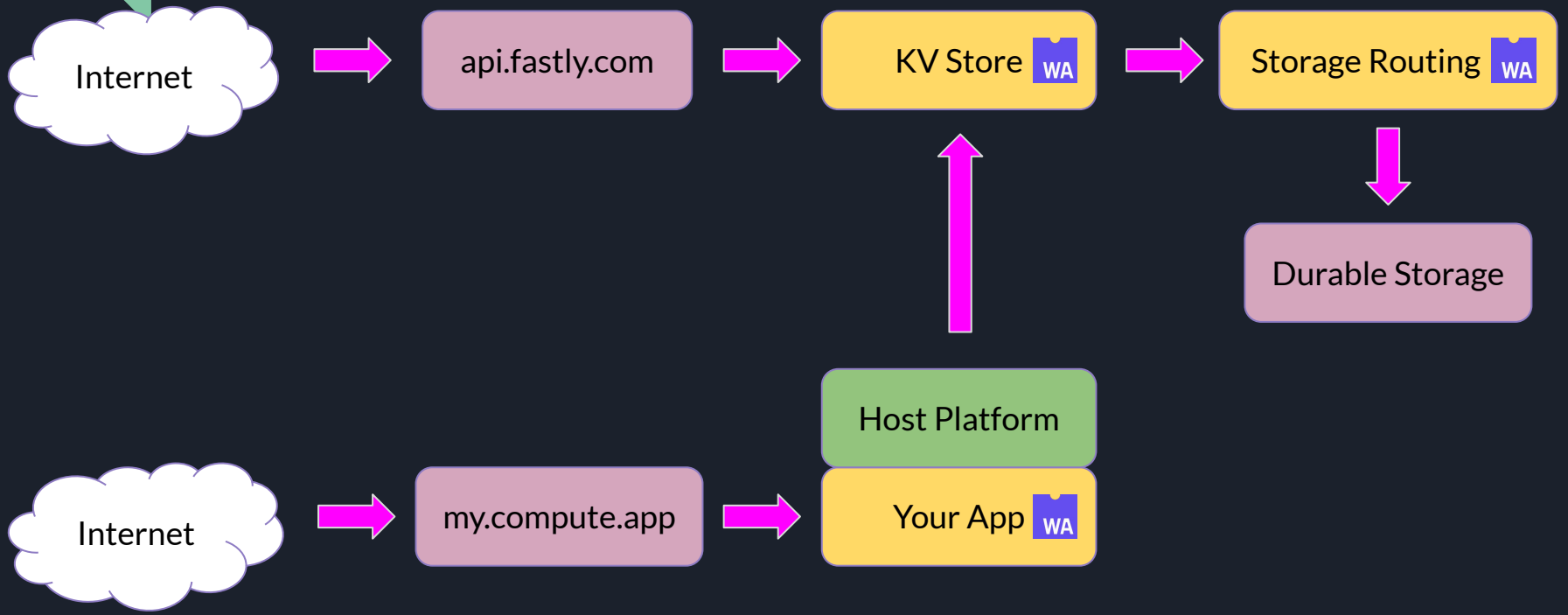
KV Store HTTP API

KV Store library API

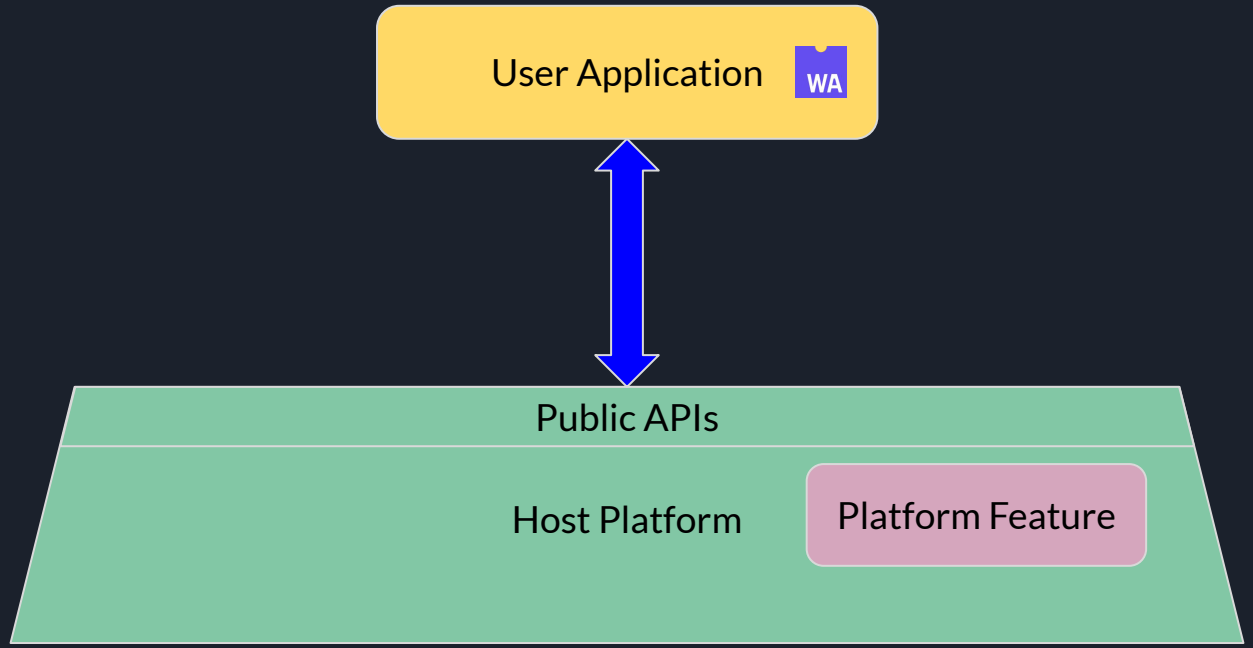
```rust

let store: KVStore = KVStore::open("mystore")?.unwrap();

let key: &str = "message1";
let value: &str = "this is message 1";

store.insert(&key, value)?;

let mut resp: LookupResponse = store.lookup(&key)?;
assert_eq!(value, resp.take_body().into_string());

store.delete(&key)?;

Ok(Response::from_status(200))

```

```
## INSERT
$ curl -X PUT /stores/kv/mystore/keys/message1 \
    -d "this is message 1"
HTTP/2 200

## LOOKUP
$ curl -X GET /stores/kv/mystore/keys/message1
HTTP/2 200
x-cache: MISS

"this is message 1"

## LOOKUP
$ curl -X GET /stores/kv/mystore/keys/message1
HTTP/2 200
x-cache: HIT

"this is message 1"

## DELETE
$ curl -X DELETE /stores/kv/mystore/keys/message1
HTTP/2 204

## LOOKUP
$ curl -X GET /stores/kv/mystore/keys/message1
HTTP/2 404
x-cache: MISS

## LOOKUP
$ curl -X GET /stores/kv/mystore/keys/message1
HTTP/2 404
x-cache: HIT
```

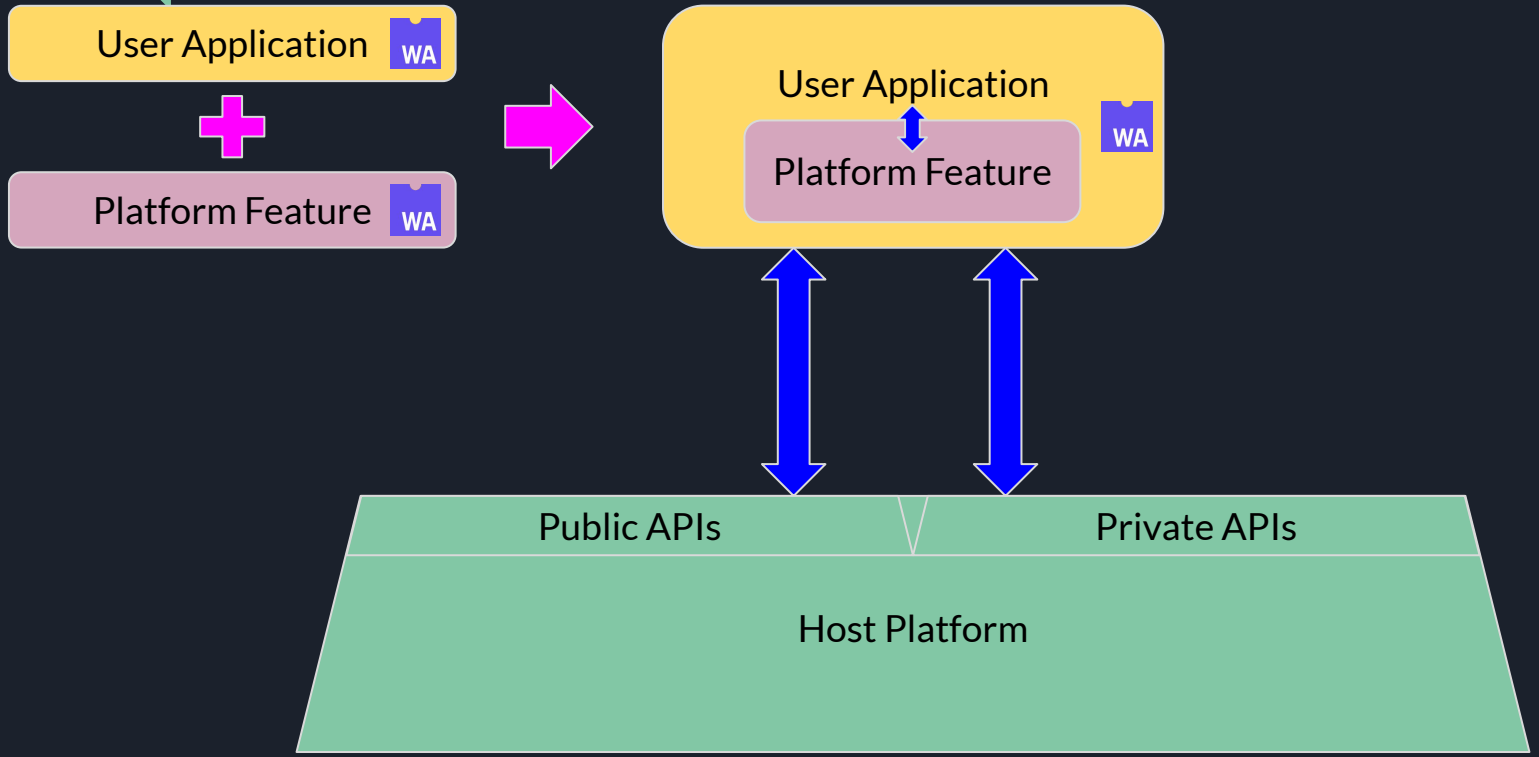
# Today's Infrastructure



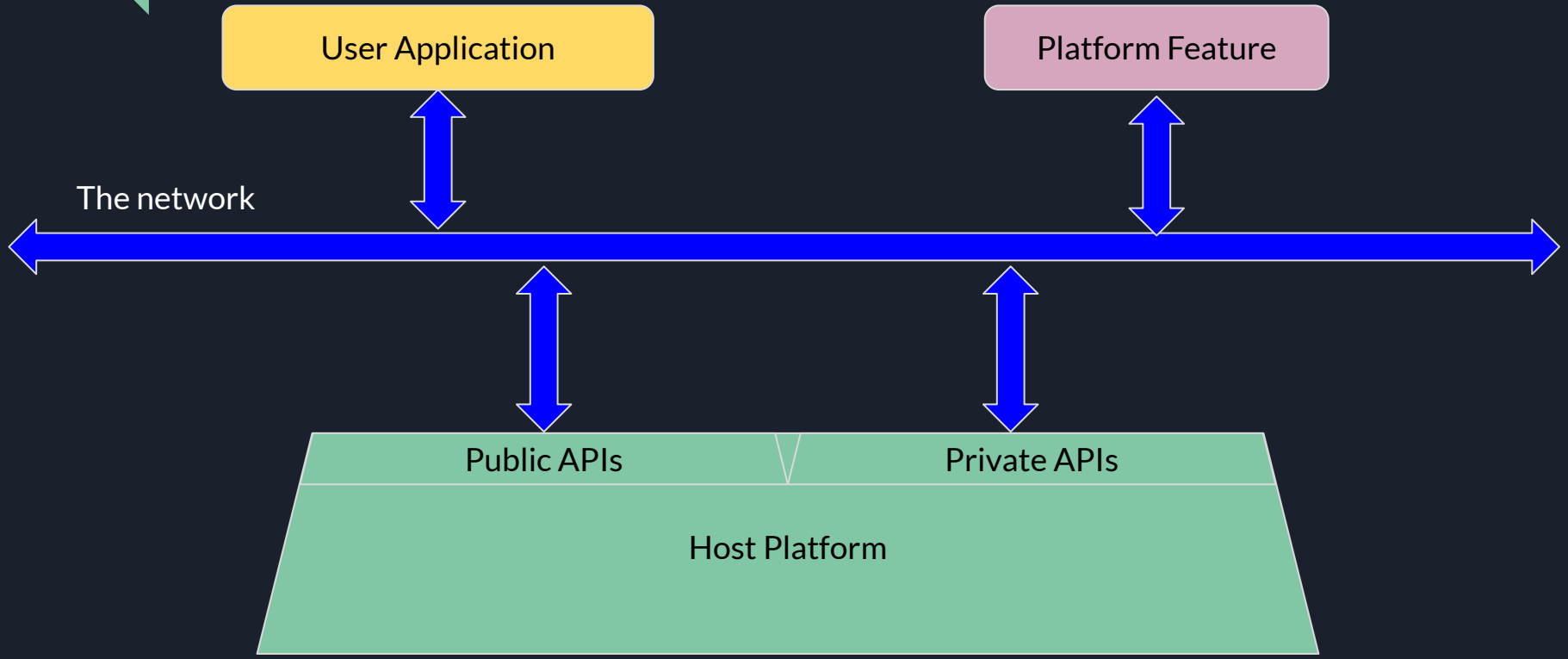
# Adding a feature to a platform



# Add the a feature as a library?



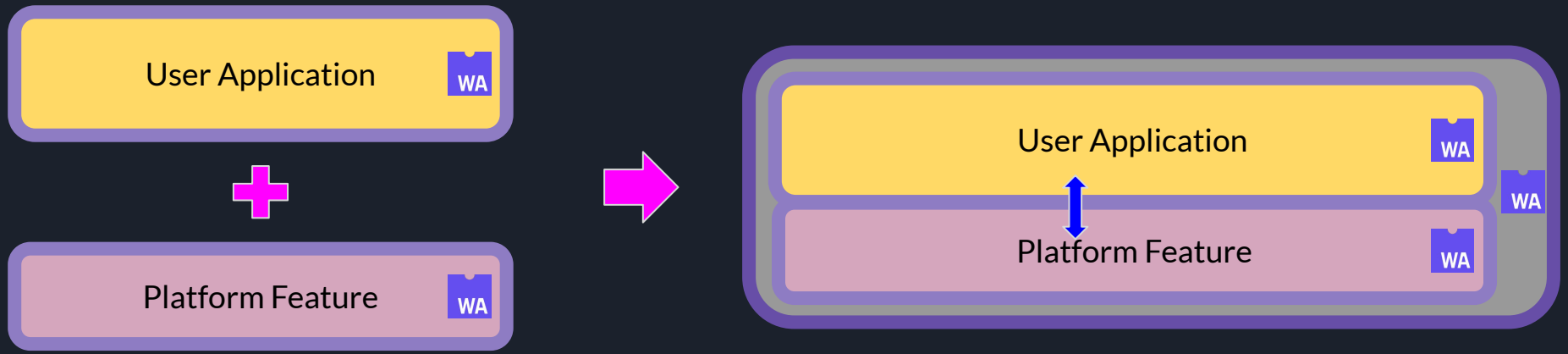
# Microservice architecture?



# Wasm Components:

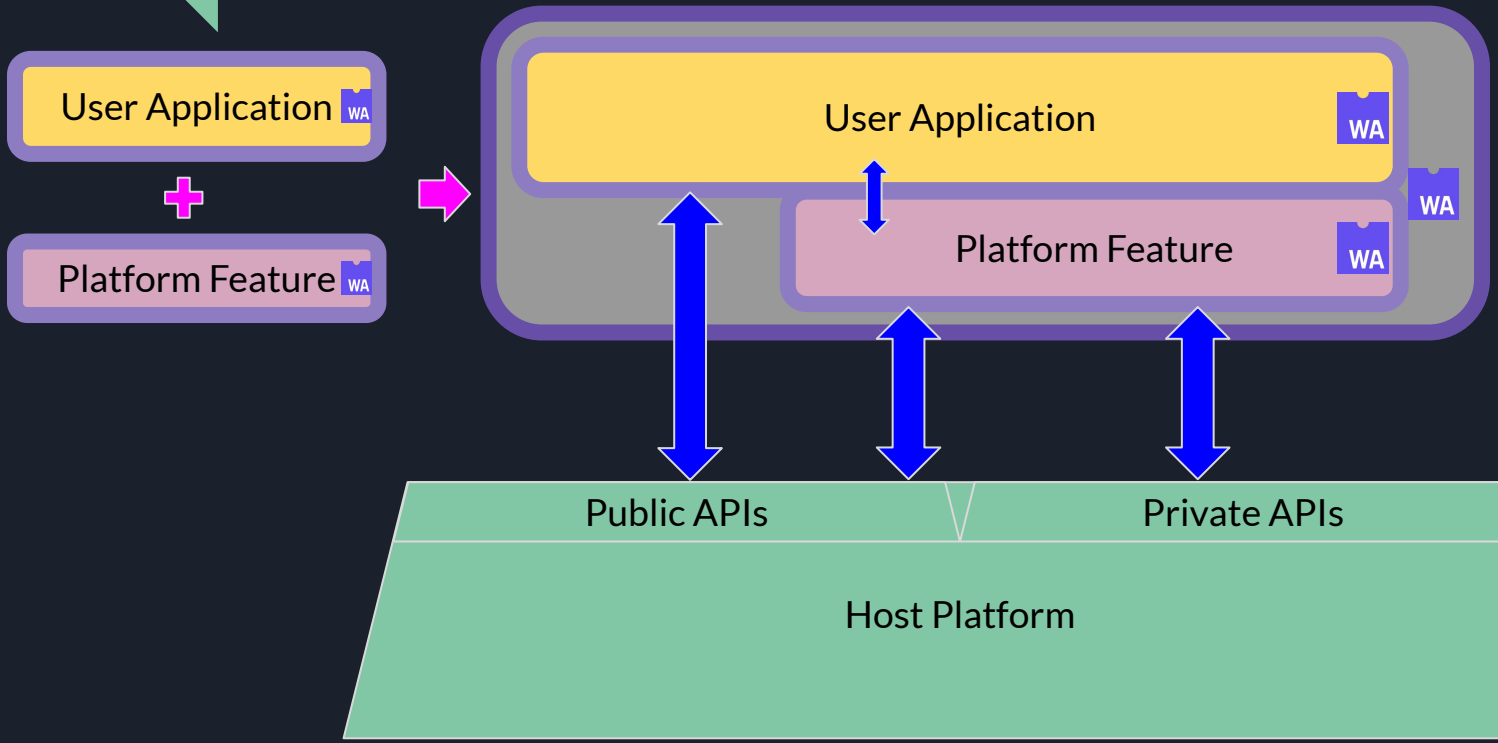
# Virtual Platform Layering

# Linking components produces a component



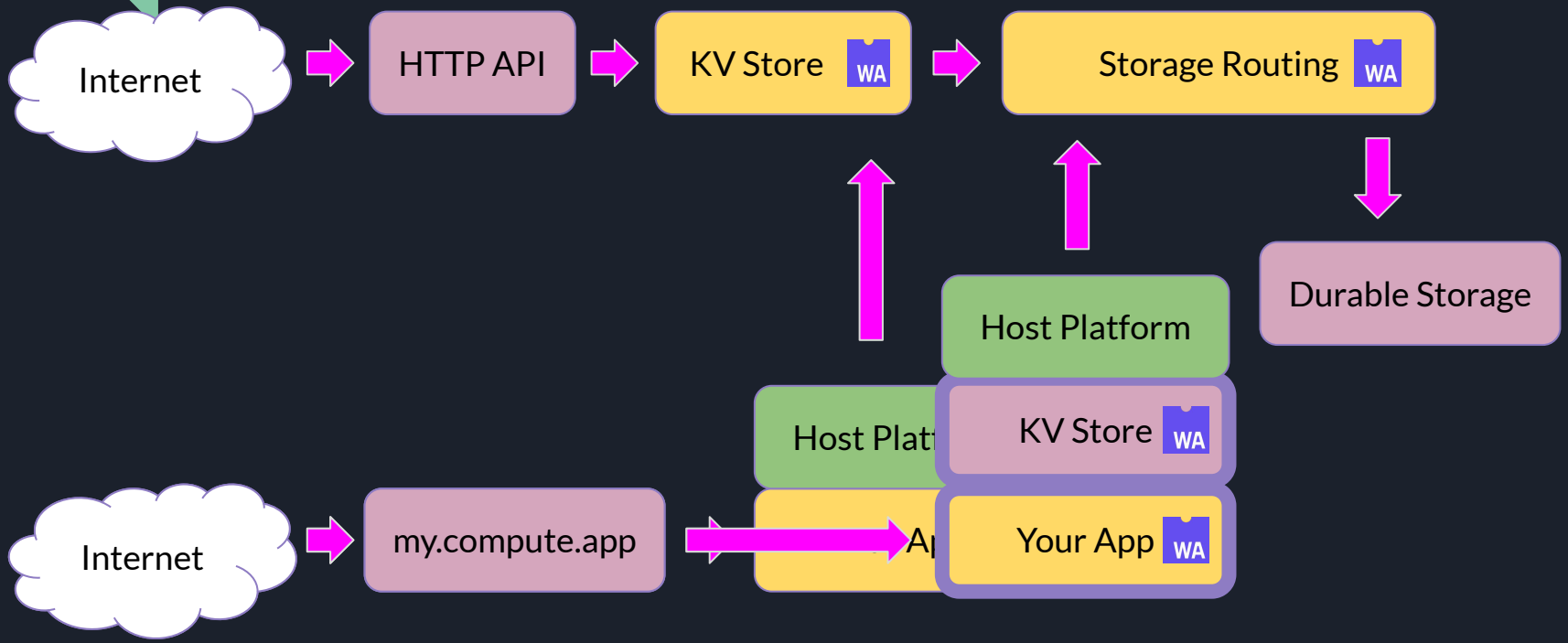

# Linking components preserves isolation

# Virtual Platform Layering!

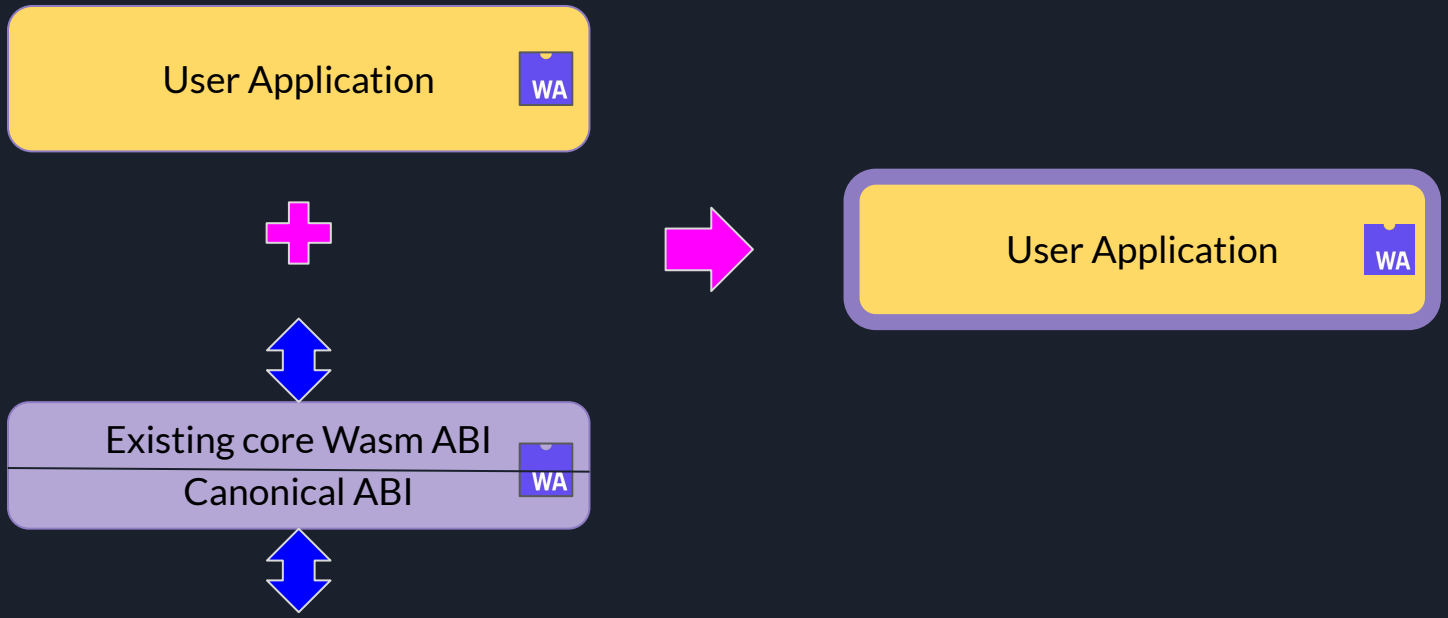



- Platform feature runs inside the Wasm sandbox
- Platform feature remains isolated from user application
- Linking, rather than routing and protocols

# Today's Infrastructure

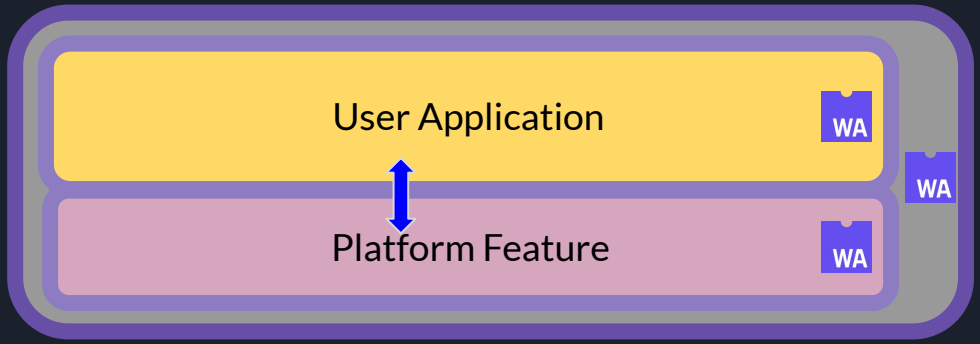


# Migrating to components



# Compiling and executing components

- Instantiate the outer component
  - Instantiate the Platform Feature component
    - Instantiate its inner core module, importing from the components' imports
  - Instantiate the User Application component
    - Instantiate its inner core modules, importing from the component's imports and the platform feature



# Developer Experience

# .witx and invocation

```
(module $fastly_kv_store
    (@interface func (export "open")
        (param $name string)
        (result $err (expected $kv_store_handle (error $fastly_status)))
    )

    (@interface func (export "lookup")
        (param $store $kv_store_handle)
        (param $key string)
        (param $lookup_config_mask $kv_lookup_config_options)
        (param $lookup_configuration (@witx pointer $kv_lookup_config))
        (param $handle_out (@witx pointer $kv_store_lookup_handle))
        (result $err (expected (error $fastly_status)))
    )

    (@interface func (export "lookup_wait_v2")
        (param $handle $kv_store_lookup_handle)
        (param $body_handle_out (@witx pointer $body_handle))
        (param $metadata_buf (@witx pointer (@witx char8)))
        (param $metadata_buf_len (@witx usize))
        (param $nwritten_out (@witx pointer (@witx usize)))
        (param $generation_out (@witx pointer u64))
        (param $kv_error_out (@witx pointer $kv_error))
        (result $err (expected (error $fastly_status)))
    )
)
```

```rust
/// Look up a value in the KV Store.
///
/// Returns `Ok(PendingLookupHandle)` if the creation of the async request was successful.
pub fn lookup(&self, key: impl AsRef<[u8]>) -> Result<PendingLookupHandle, KVStoreError> {
    let mut pending_lookup_handle_out = INVALID_KV_PENDING_LOOKUP_HANDLE;
    let key = key.as_ref();
    let config_options = LookupConfigOptions::empty();
    let config = LookupConfig::default();
    let status = unsafe {
        sys::lookup_v2(
            self.as_u32(),
            key.as_ptr(),
            key.len(),
            config_options,
            &config,
            &mut pending_lookup_handle_out,
        )
    };
    status.result().map_err(|st| match st {
        FastlyStatus::BADF => KVStoreError::InvalidStoreHandle,
        FastlyStatus::INVAL => KVStoreError::ItemBadRequest,
        _ => st.into(),
    })?;
    if pending_lookup_handle_out == INVALID_KV_PENDING_LOOKUP_HANDLE {
        Err(KVStoreError::Unexpected(FastlyStatus::ERROR))
    } else {
        Ok(unsafe { PendingLookupHandle::from_u32(pending_lookup_handle_out) })
    }
}
```

# .wit and invocation

```
resource lookup-response {
  take-body: func() -> body-handle;
  try-take-body: func() -> option<body-handle>;
  take-body-bytes: func() -> list<u8>;
  metadata: func() -> option<list<u8>>;
  generation: func() -> u64;
}

open: func(name: list<u8>) -> result<store, error>;

resource store {
  lookup: func(
    key: list<u8>,
  ) -> result<lookup-response, error>;

  build-lookup: func() -> lookup-builder;
}
```

```
let store: Store = kv_store::open(name: b"kv1")?;
let value: LookupResponse = store.lookup(key: b"msg3")?;
```

# .witx and optional parameters

```
(typename $kv_insert_config_options
    (flags (@witx repr u32)
       $reserved
       $background_fetch
       ;; reserved_2 was previously if_generation_match (u32)
       $reserved_2
       $metadata
       $time_to_live_sec
       $if_generation_match
       ))

(typename $kv_insert_mode
    (enum (@witx tag u32)
       $overwrite
       $add
       $append
       $prepend))

(typename $kv_insert_config
  (record
    (field $mode $kv_insert_mode)
    (field $unused u32)
    (field $metadata (@witx pointer (@witx char8)))
    (field $metadata_len u32)
    (field $time_to_live_sec u32)
    (field $if_generation_match u64)
    ))
```

```
let mut config_options = InsertConfigOptions::empty();
let mut config = InsertConfig::default();

config.mode = mode;

if background_fetch {
    config_options.insert(InsertConfigOptions::BACKGROUND_FETCH);
}

if let Some(igm) = if_generation_match {
    config.if_generation_match = igm;
    config_options.insert(InsertConfigOptions::IF_GENERATION_MATCH);
}

if !metadata.is_empty() {
    config.metadata = metadata.as_ptr();
    config.metadata_len = metadata.len() as u32;
    config_options.insert(InsertConfigOptions::METADATA);
}

if let Some(ttl) = time_to_live_sec {
    config.time_to_live_sec = ttl.as_secs().try_into().unwrap_or_default();
    config_options.insert(InsertConfigOptions::TIME_TO_LIVE_SEC);
}

let mut pending_insert_handle_out = INVALID_KV_PENDING_INSERT_HANDLE;

let status = unsafe {
    sys::insert_v2(
```

# .wit and optional parameters

```
resource insert-builder {
  mode: func(mode: insert-mode);
  background-fetch: func();
  if-generation-match: func(generation: u64);
  metadata: func(data: list<u8>);
  time-to-live: func(ttl: u64);

  execute: func(
    key: list<u8>,
    value: list<u8>,
  ) -> result<_, error>;

  execute-async: func(
    key: list<u8>,
    value: list<u8>,
  ) -> result<pending-insert, error>;
}
```

```
let store: Store = kv_store::open(name: b"kv1")?;
let insert: InsertBuilder = store.build_insert();
insert.mode(Append);
insert.background_fetch();
insert.metadata(b"version: 1");
insert.time_to_live(300_000);
insert.if_generation_match(65535);

let result: () = insert.execute(key: b"my_key", value: b"my_value")?;
```

# What can’t wit do?

```rust
let store: KVStore = KVStore::open("kv1")?.unwrap();
store.build_insert()
    .mode(Append)
    .background_fetch()
    .metadata("version: 1")
    .time_to_live(Duration::from_secs(300))
    .if_generation_match(65535)
    .execute("my_key", "my_value")?;
```

```rust
let store: Store = kv_store::open(name: b"kv1")?;
let insert: InsertBuilder = store.build_insert();
insert.mode(Append);
insert.background_fetch();
insert.metadata(b"version: 1");
insert.time_to_live(300_000);
insert.if_generation_match(65535);

let result: () = insert.execute(key: b"my_key", value: b"my_value")?;
```

# Developer Experience: 👍

Search this site...

Working Groups

Cloud Native AI Working Group

WebAssembly (WASM) Working Group

WebAssembly (Wasm) Working Group Charter

WASM Working Group Deliverables

Wasm OCI Artifact layout

Batch System Initiative Working Group

Container Orchestrated Device Working Group

# Wasm OCI Artifact layout

## Wasm OCI Artifact layout

This is a guide for an **OCI image format** for Wasm that can be used across projects in order to enable consistency for users, tooling and registries. This OCI artifact has a specific Wasm `config.mediaType` of `application/vnd.wasm.config.v0+json` to designate it as an **OCI artifact**. It contains a representation of Wasm configuration that can be broadly be used across projects (various Wasm runtimes and container runtimes). It includes the ability to include wasm modules or components in the artifact.

It is designed to be compliant with the artifact guidance of the **OCI image spec** in order to be compatible with the widest range of registries today.

### Goals

- Single OCI compliant artifact type that can be used across container runtimes and Wasm runtime projects (wasmcloud, spin, containerd, podman, etc)
- Single layer type that is supported where everything is a component, which makes it so all the tools don't need to know about all possible layer types (eg. static files, Wasm runtime config files, etc.).
- Should be able to support an "exploded" representation of the component where each component is a layer and can be combined ("composed") by either Wasm runtime or container runtime
- Have fall back if registries don't support artifacts (via config.media type)

# An Edge KV Store built with Wasm Components