KubeCon

CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024

KubeCon

CloudNativeCon

China 2024

Expanding Cloud Native Capabilities with WASM

A Case Study of Harbor and WASM Integration

Chenyu Zhang, AntGroup
Yan Wang, Broadcom

Agenda

- Harbor Intro
- WASM Intro
- Challenges
- Case Study
- Summary & Future Outlook
- Q & A

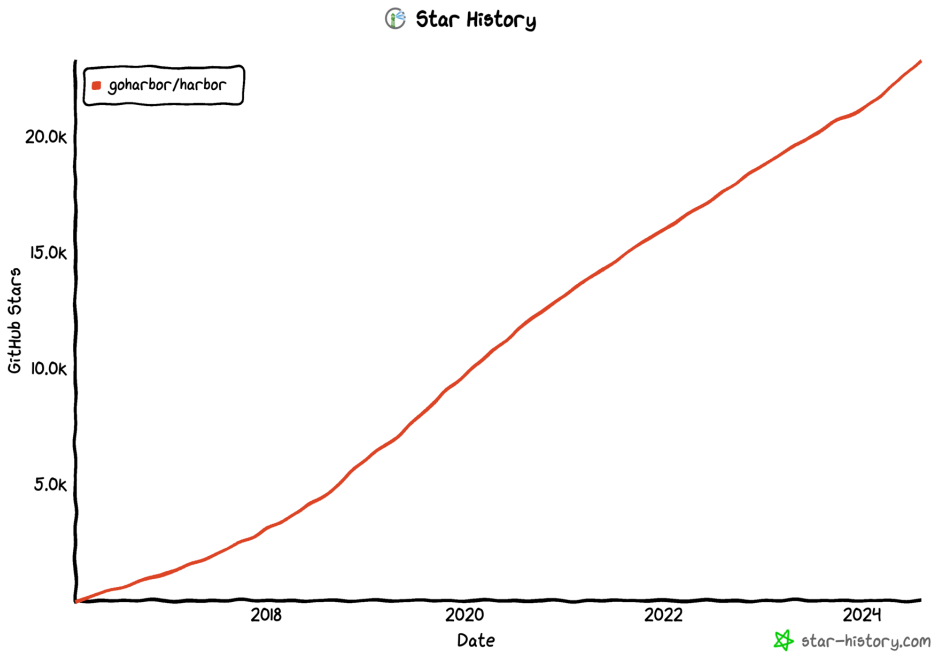
Harbor Intro

HARBOR

What is Harbor?

Harbor is an open source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted. Harbor, a CNCF Graduated project, delivers compliance, performance, and interoperability to help you consistently and securely manage artifacts across cloud native compute platforms like Kubernetes and Docker.

https://goharbor.io



Core Features

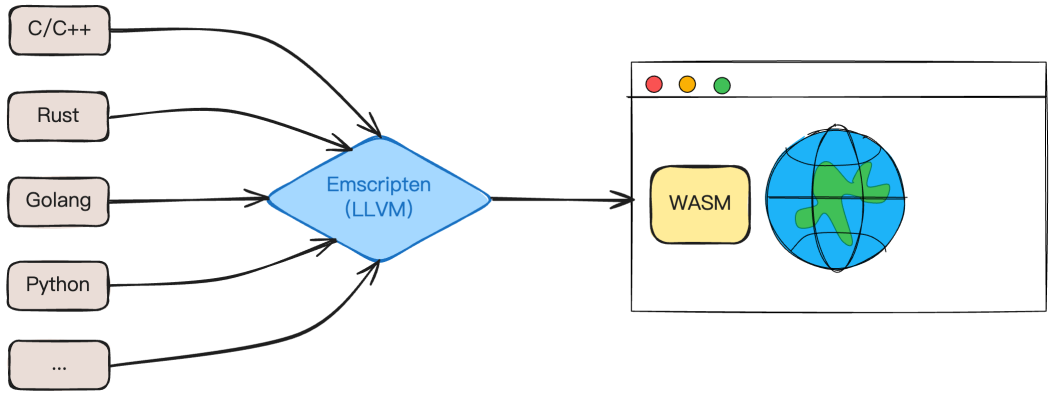
- Access Control
  - ✓ RBAC, Project Isolation
- Artifact Distribution
  - ✓ Replication, Proxy Cache, P2P Preheat
- Security & Compliance
  - ✓ Vulnerability Scan, Signature, CVE Export, Security Hub
- Policy & Maintainability
  - ✓ Quota, Immutability, Retention, Garbage Collection, Log Rotation
- Extensibility
  - ✓ OIDC/LDAP Auth, Webhook, Pluggable Scanner, Robot Account

Architecture

What is WASM?

WebAssembly (wasm) is a code compilation format that enables programs written in high-level languages to run in web browsers and has the characteristic of near-native execution speed. It is not a programming language but a compilation target compatible with various programming languages. By allowing developers to compile code written in languages such as C/C++, Rust, etc. into binary instructions and run them in a sandbox environment, it has achieved the ability to perform complex computing and high-performance tasks on the web page.

Originally designed for the web, it is now used in a wide range of applications beyond the browser.



Benefits of WASM

Performance

- WASM is designed to be fast, with near-native execution speed.
- Efficient compilation and execution on modern hardware.

Portability

- Platform-independent, can run on any hardware with a WASM runtime.
- Compatible with various languages (e.g., C, C++, Rust, Go).

Security

- Sandboxed execution environment, ensuring safety and security.
- Prevents arbitrary code execution, reducing security risks.

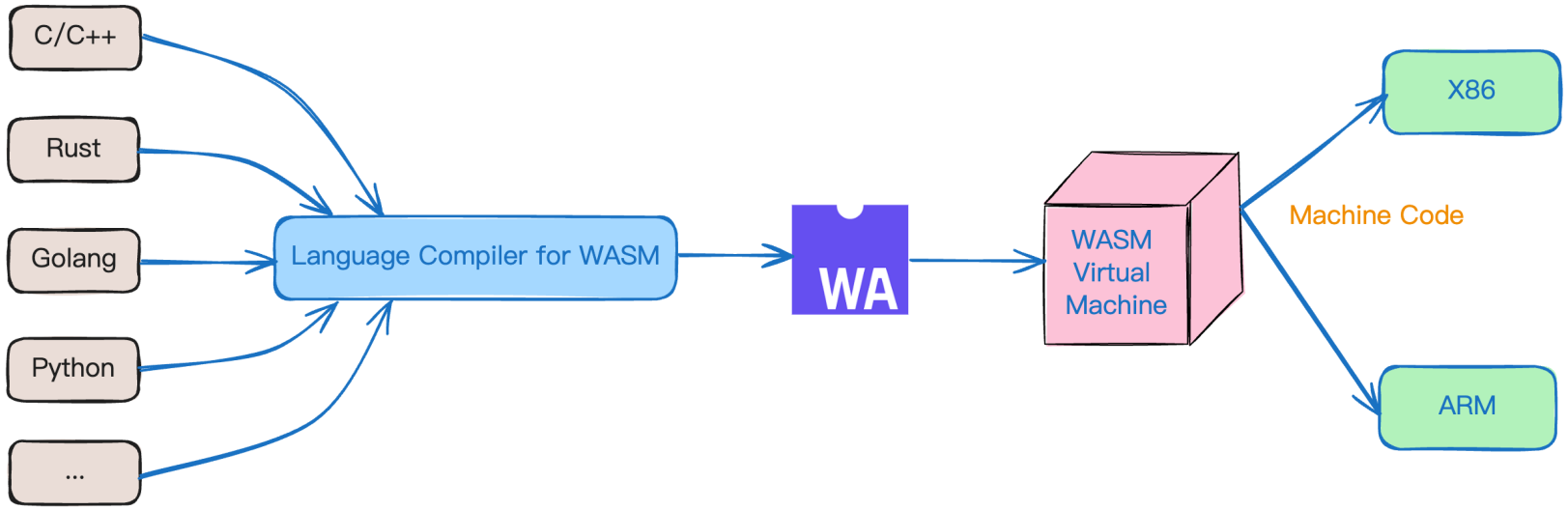
Interoperability

- Seamless integration with existing web technologies (e.g., JavaScript).
- Can be used alongside other technologies in a hybrid application.

WASI - run WASM everywhere

WebAssembly System Interface (WASI) is a standard API designed for WebAssembly.

It allows WebAssembly modules to perform system-level tasks, such as file I/O and network communication, in a secure and platform-independent manner.



Current Situation

Although Harbor already provides some extension way for user, but there are some limitations as follows:

Intrusiveness

- Integrate new features or functionalities involves modifications to the codebase, although harbor provides the interface to simplify implementations, but it may disrupt the stability of the original application.

Complexity

- Require knowledge of the underlying system and dependencies.
- Additional configurations may need to be introduced which may leading to increased configuration overhead.
- Increased complexity and maintenance costs for later changes.

Uncontrollability

- Although user can configure webhook to watch the events for pull/push or other type events happened, but webhook can only passively receive notifications and can not take actions on them.

Idea inspired from eBPF

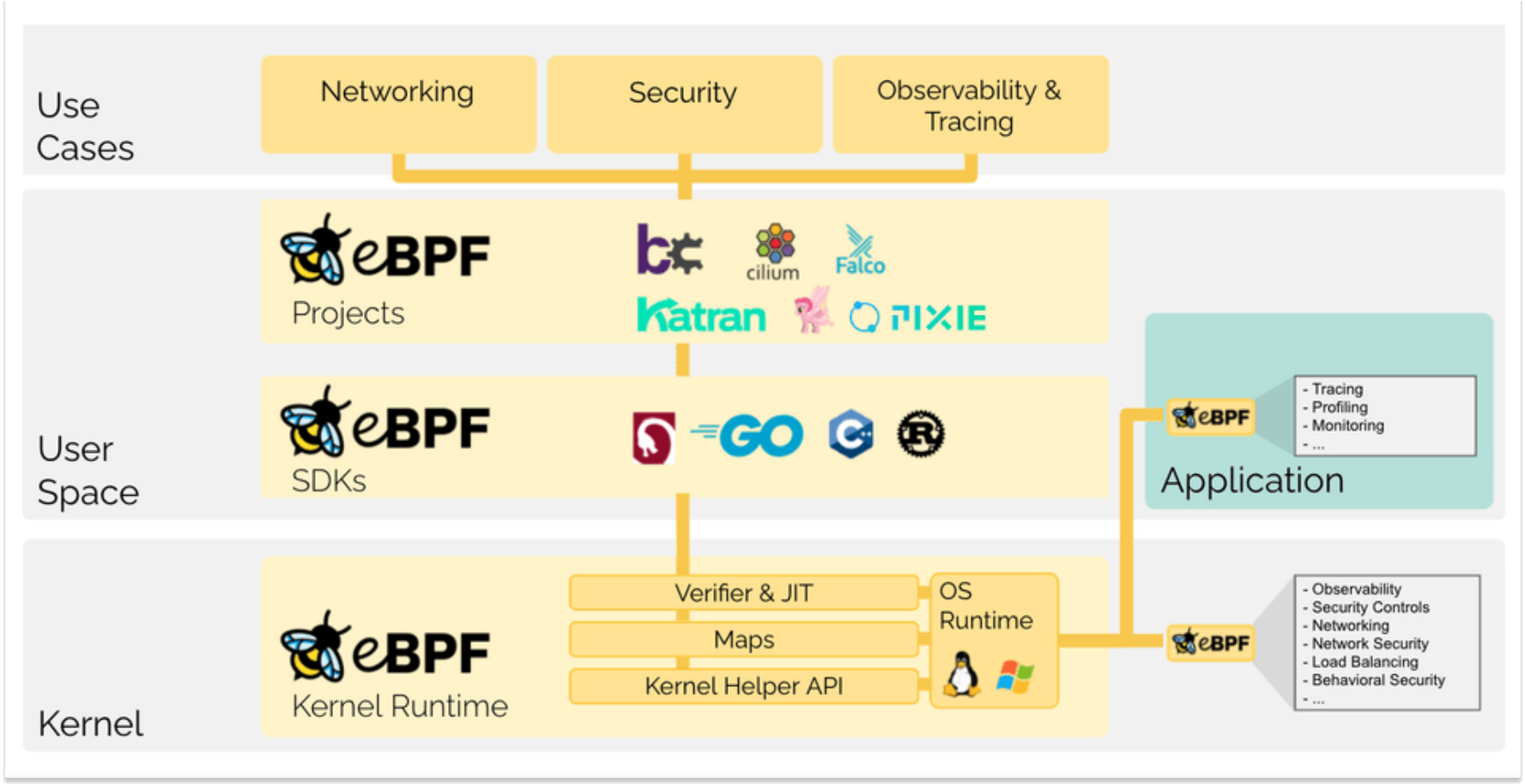
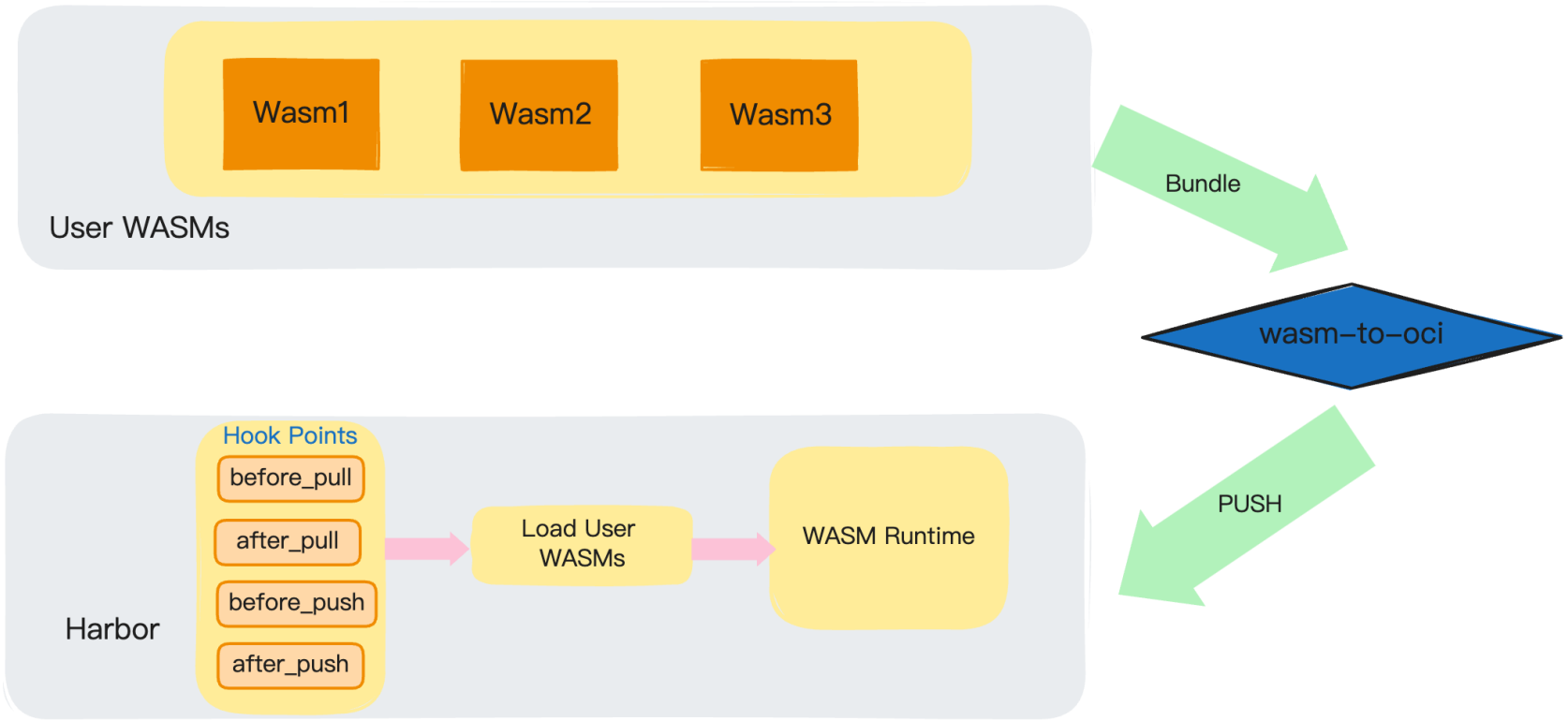


Image reference: https://ebpf.io/

Harbor + WASM



Case Study

Try WASM

main.rs

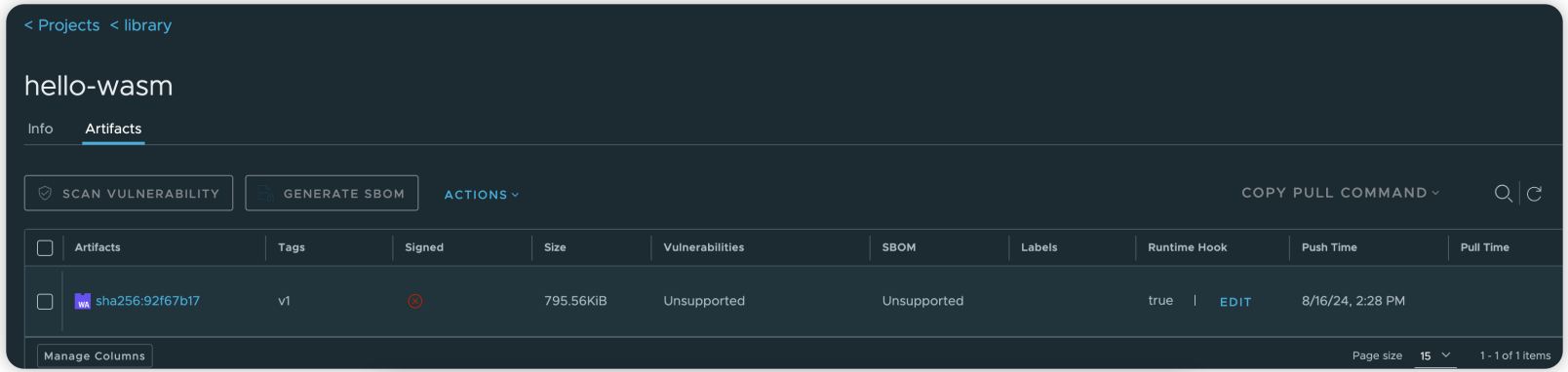
```rust
fn main() {
    println!("Hello, WASM!")
}
```

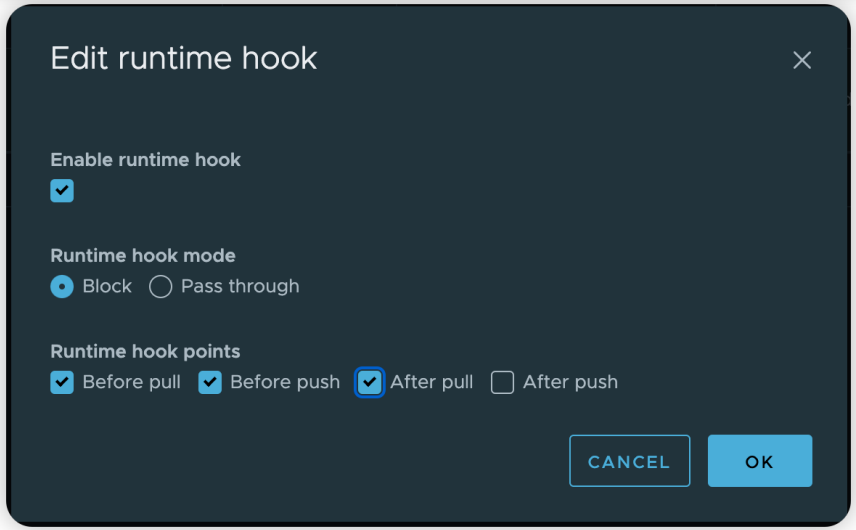
main.go

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello WASM!")
}
```

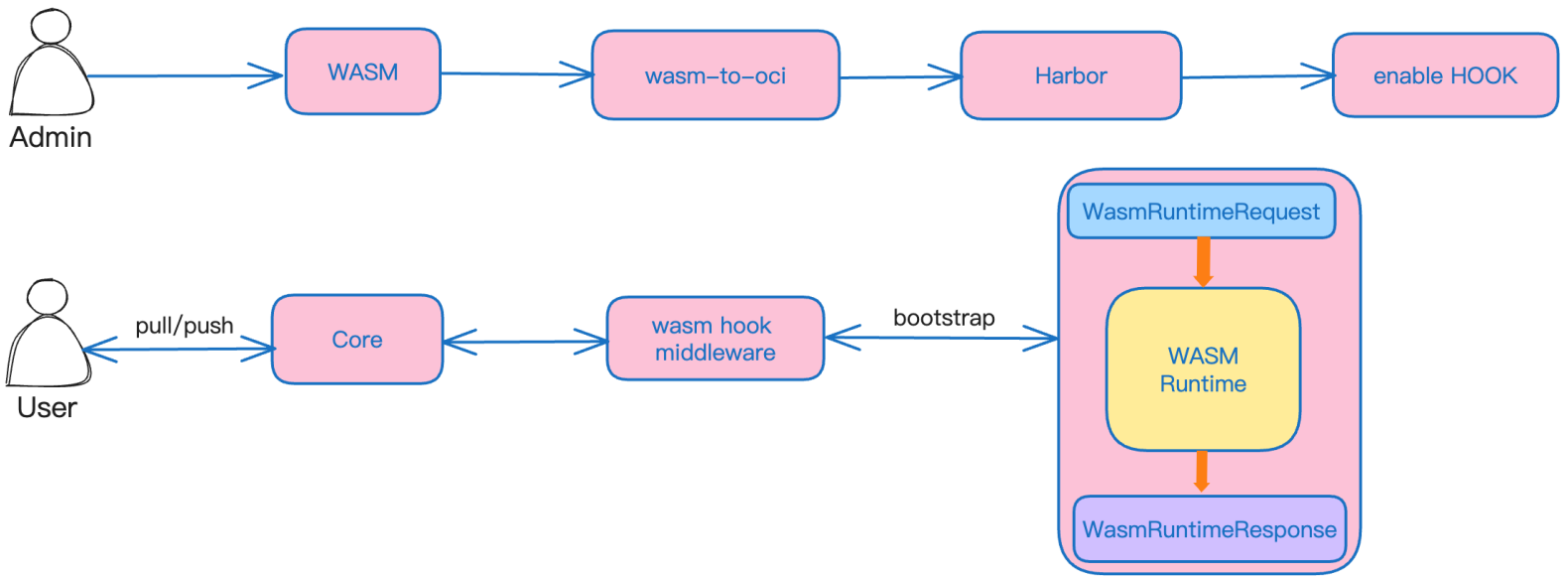
```
$ rustup target add wasm32-wasip1
$ rustc main.rs -o hello.wasm --target wasm32-wasip1
$ wasmtime hello.wasm
Hello, WASM!

$ GOOS=wasip1 GOARCH=wasm go build -o hello.wasm main.go
$ wasmtime hello.wasm
Hello, WASM!
```

Harbor integration

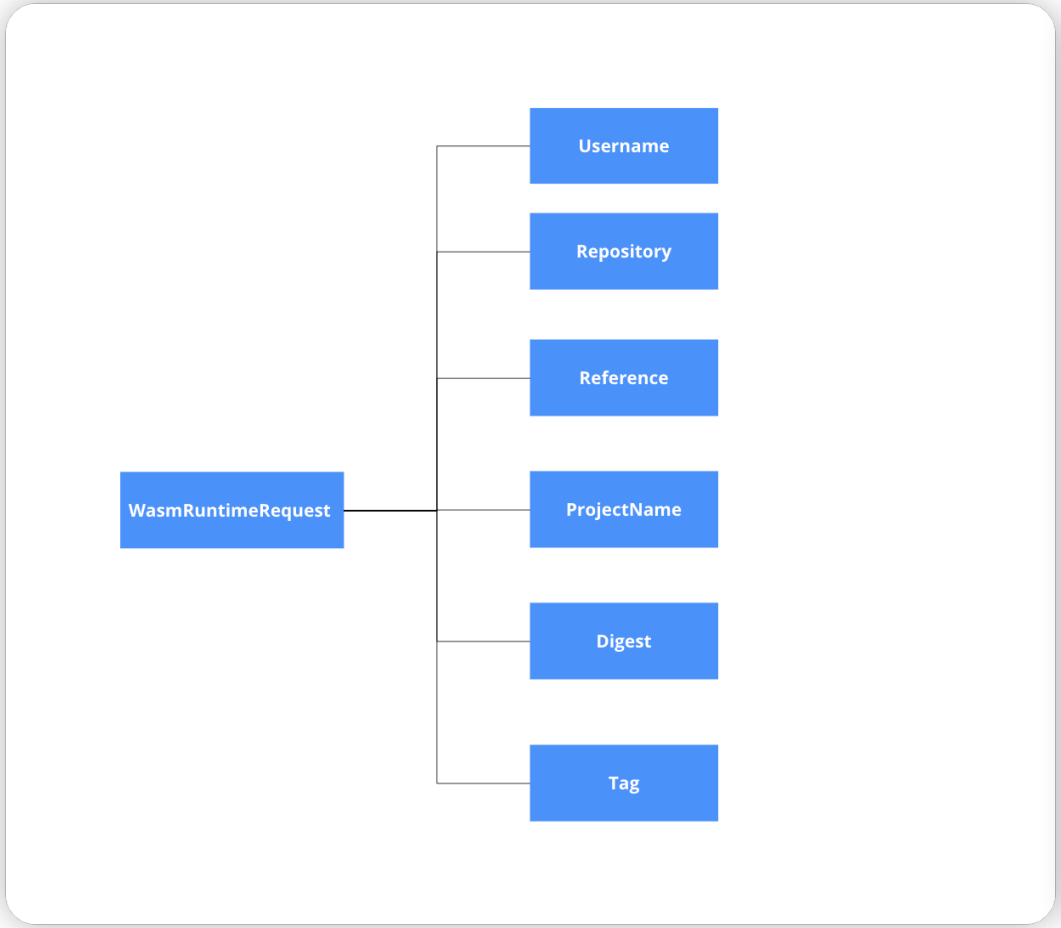


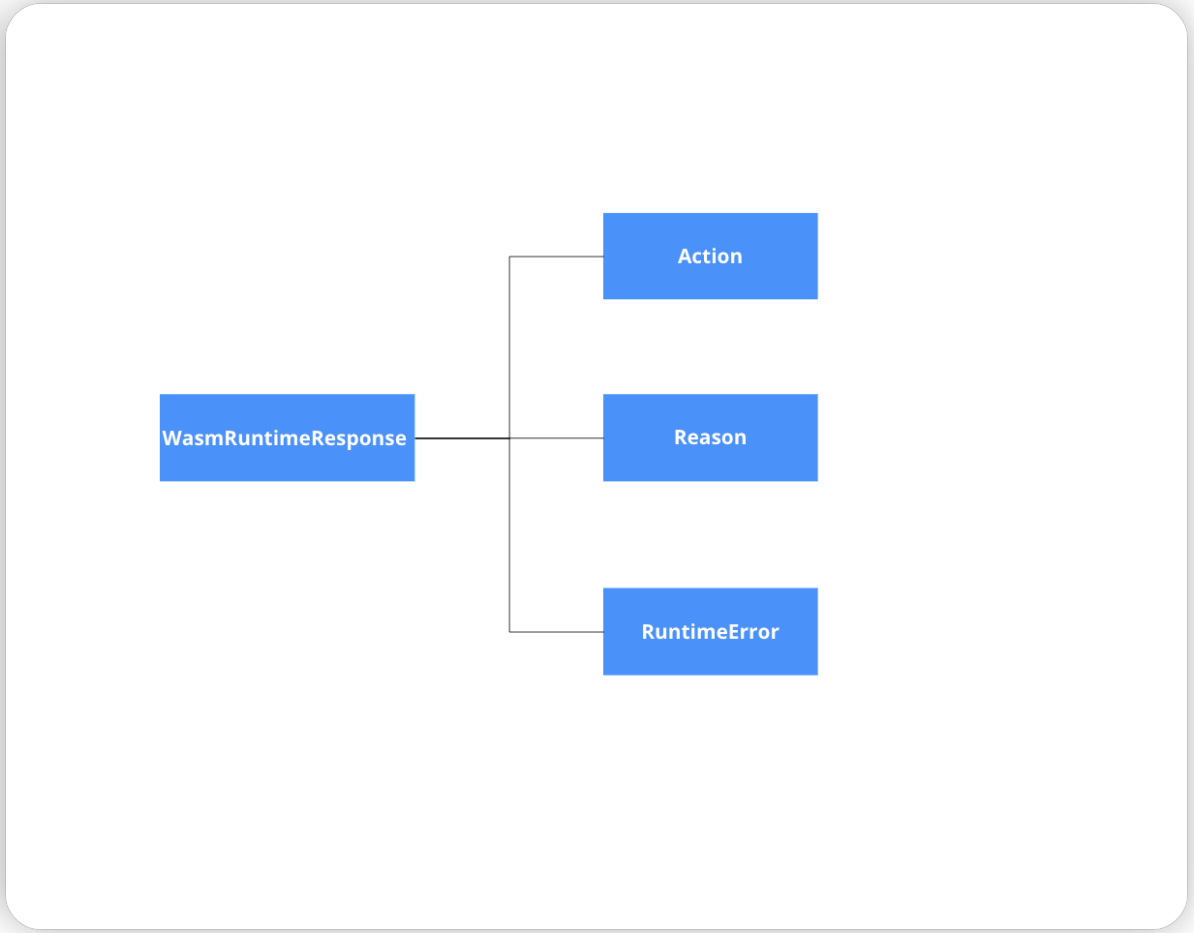


Workflow



WASM Input & Output Model





Case Study

Example

A WASM implemented by golang which does not allow user alice to pull the artifact library/alpine.

wasm.go

```go
package main

// tinygo build -o hello.wasm -scheduler=none -target=wasi main.go
// main is required for TinyGo to compile to Wasm.
func main() {}

const (
    ActionAllow = "allow"
    ActionDeny = "deny"
)

//export before_pull
func before_pull(req *RuntimeRequest) *RuntimeResponse {
    resp := &RuntimeResponse{}
    // your business logic:
    // here we do not allow alice to pull the library/alpine
    if req.Username == "Alice" && req.Repository == "library/alpine" {
        resp.Action = ActionDeny
        resp.Reason = "alice not be allowed to pull this image"
        return resp
    }

    resp.Action = ActionAllow
    return resp
}
```

Demo Time

Benefits from Harbor+WASM

Complexity

- Users can use their own preferred programming languages to develop WASM programs.
- Users do not need the knowledge of harbor's codebase.

Flexibility

- WASM packaged in OCI format can be stored and managed natively on harbor.
- The ability to expand the harbor capabilities in a non-invasive way.
- Support dynamic updates of WASM program without requiring a harbor restart.
- Users have the flexibility to plug/unplug and combine different WASMs to achieve desired outcomes.

Maintainability

- By keeping WASM programs separate from the harbor codebase, post-maintenance costs are reduced.
- Generic WASM can be shared within the community, enhancing development efficiency.

Future Outlook

The current implementation is just a demo level experiment, and there are still a lot of things that need to be improved if we want to really introduce this feature to harbor.

Management

- Unified management center to manage the WASM lifecycle.
- Policy-driven, applying rules to match target artifacts and actions taken.
- Visualize WASM execution records and results to facilitate troubleshooting.

Performance

- Bootstrap the WASM runtime and running WASM programs at hooks may affect performance.
- A bad WASM program might take up much harbor’s CPU and memory resources.

Security

- Need better lifecycle management for WASM runtime.
- A better isolation mechanism is needed between the WASM runtime and the operation of the harbor's own components

Q & A

Thank You!