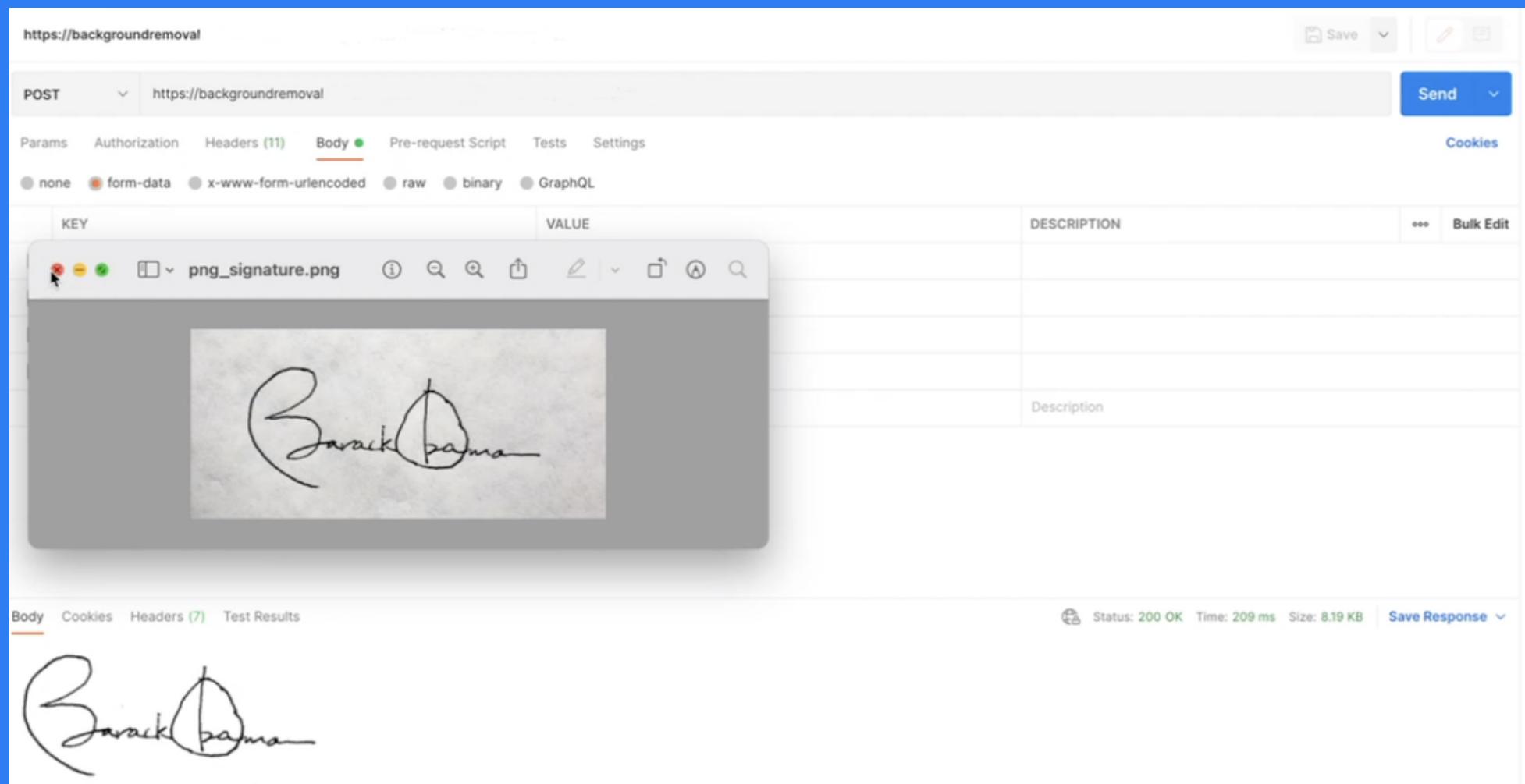


A Kubernetes and Wasm Case Study

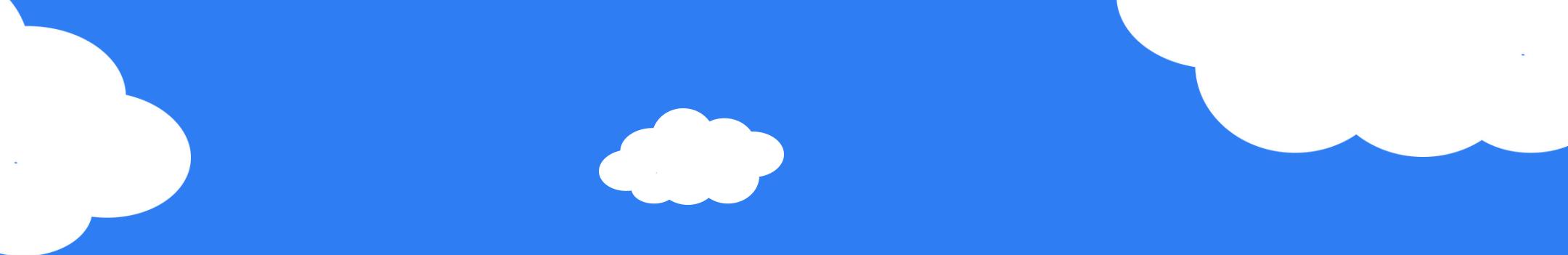
Use Case 1: Running Individual Functions in wasmCloud



The screenshot shows a Postman request to `https://backgroundremoval`. The method is set to `POST`, and the URL is also `https://backgroundremoval`. In the `Body` tab, there is a file attachment named `png_signature.png` containing a handwritten signature of "Barack Obama". The response status is `200 OK` with a time of `209 ms` and a size of `8.19 KB`.



CLOUD NATIVE
COMPUTING FOUNDATION



Here, I take an existing function that removes the background from images, translating it into Rust, and porting it to `wasmCloud`.

In the previous slide, you can see a) the original version, and b) compiled to Wasm and running as a `wasmCloud Actor`.

The screenshot shows a POST request to `https://actor-wasmcloud.corp.ethos`. The request details are as follows:

- Method: POST
- URL: `https://actor-wasmcloud.corp.ethos`
- Params tab is selected.
- Headers (8) tab is present.
- Body tab is selected (indicated by a green dot).
- Pre-request Script, Tests, and Settings tabs are present.
- Cookies tab is present.
- Send button is visible.

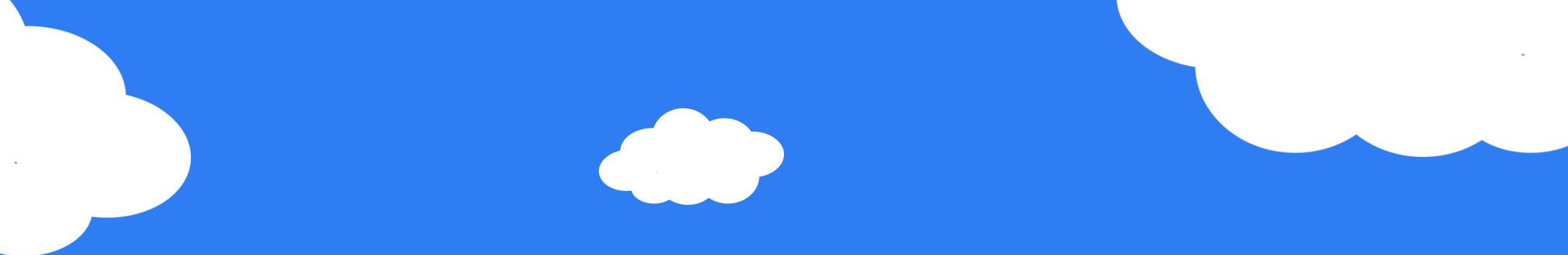
The Body section shows the response content:

373 x 170

The status bar at the bottom indicates:

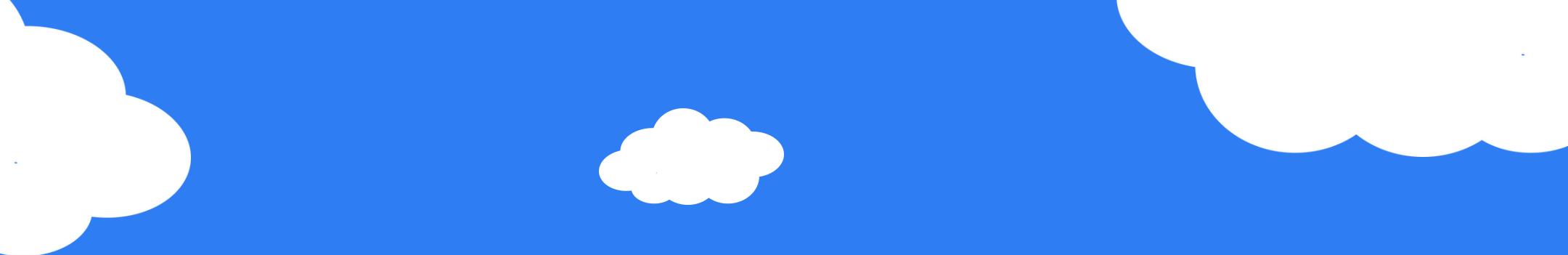
- Status: 200 OK
- Time: 201 ms
- Size: 3.82 KB
- Save Response button

Here, we can spin up an actor instantaneously to service the request, with an extremely low memory footprint versus running a service that needs a level of always-on, baseline resources.



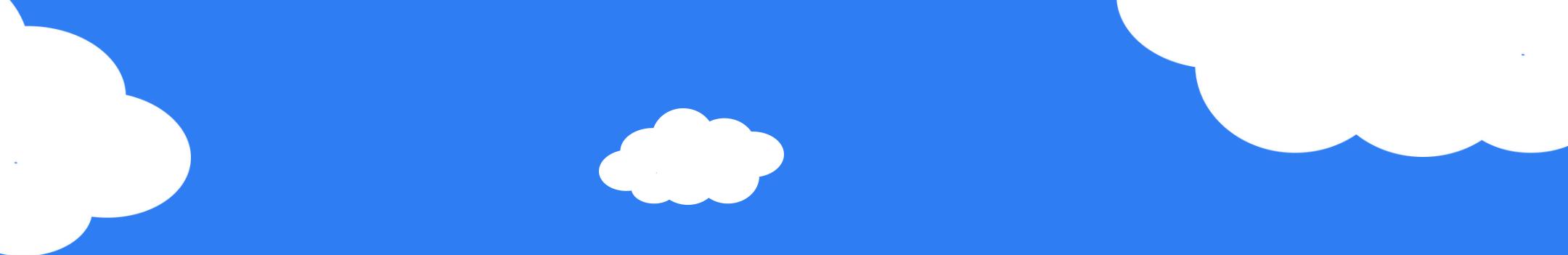
**Functions can easily be spun off
to work on-device or in the
cloud.**

This enables extremely efficient
use of cloud resources while
promoting code and module
reuse across a variety of
compute architectures.



Use Case 2: Running wasmCloud as a Service in Kubernetes Clusters

Having seen some success in porting individual functions to run as actors `wasmCloud`, I wanted to see if I could take a full microservice, currently running in Kubernetes, and make it run in Wasm. Here's the step-by-step:

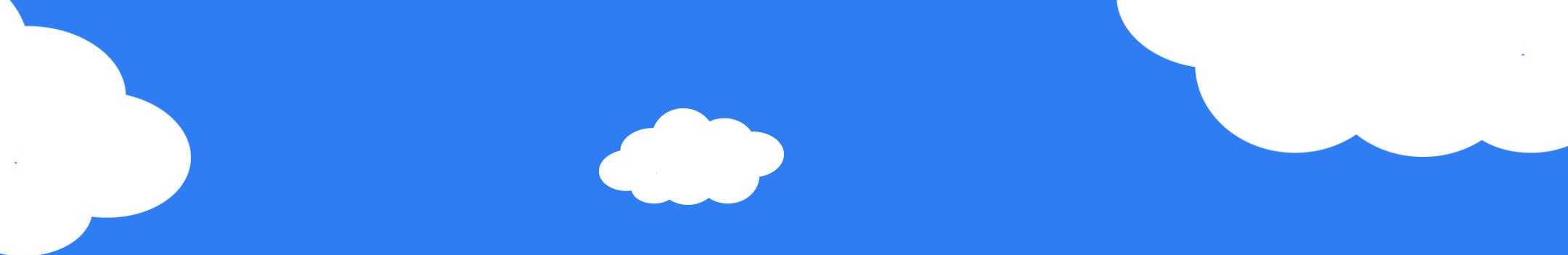


Ensure you have Helm installed, and valid credentials for a Kubernetes cluster. For me, that entails configuring kubectl to use an authenticated kubeconfig.yaml.

Normally, you can follow the single step in the documentation for installing wasmCloud via Helm.

However, I want to slightly customize this to enable the Kubernetes Applier to bootstrap application deployment. First, create a custom values.yaml:

```
wasmcloud:  
  enableApplierSupport: true  
  customLabels:  
    wasmcloud.dev/route-to: "true"  
  ...
```



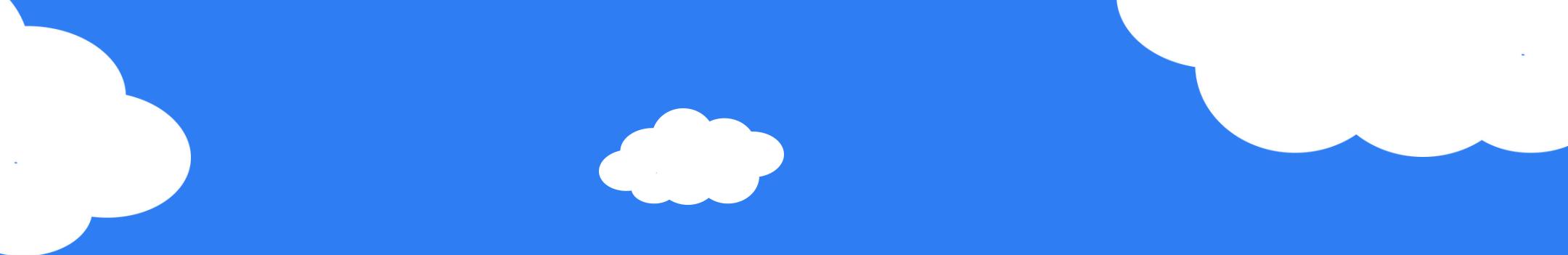
Then you can deploy via helm (with your choice of RELEASE_NAME)

```
$ helm install <RELEASE_NAME> wasmcloud/wasmcloud-host -f values.yaml
```

Finally, to verify the install, you can grab the name of the deployment:

```
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
wasmcloud-test-wasmcloud-host	1/1	1	1	4m32s



And then forward port 4000 to localhost:

```
$ kubectl port-forward deployment/wasmcloud-test-wasmcloud-host 4000  
Forwarding from 127.0.0.1:4000 -> 4000  
Forwarding from [::1]:4000 -> 4000  
Handling connection for 4000
```

Then finally simply visit localhost:4000 in your favorite browser, and you can see the wasmCloud dashboard ready to run your first apps, all from an existing Kubernetes namespace.

In Summary:

A major advantage of WebAssembly on the backend is that it can securely enable high performance and efficiency, while still being compatible with Kubernetes.

So, in a case like this, where we have huge investments in Kubernetes operations, compliance, and automation, we can integrate WebAssembly directly into our existing infrastructure.