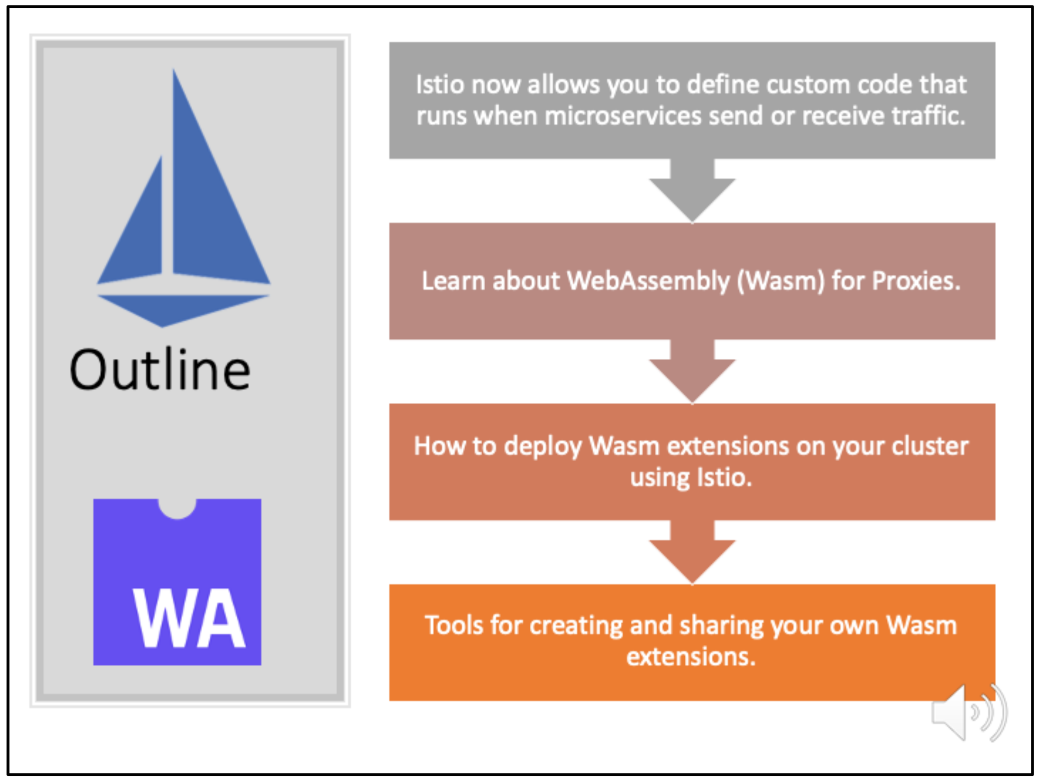
Hello, my name is Ed Snible and I will present on WebAssembly for Istio today.

Thank you Iris for hosting this event.

Also thanks to Mohammad Banikazemi for an earlier version of this presentation.

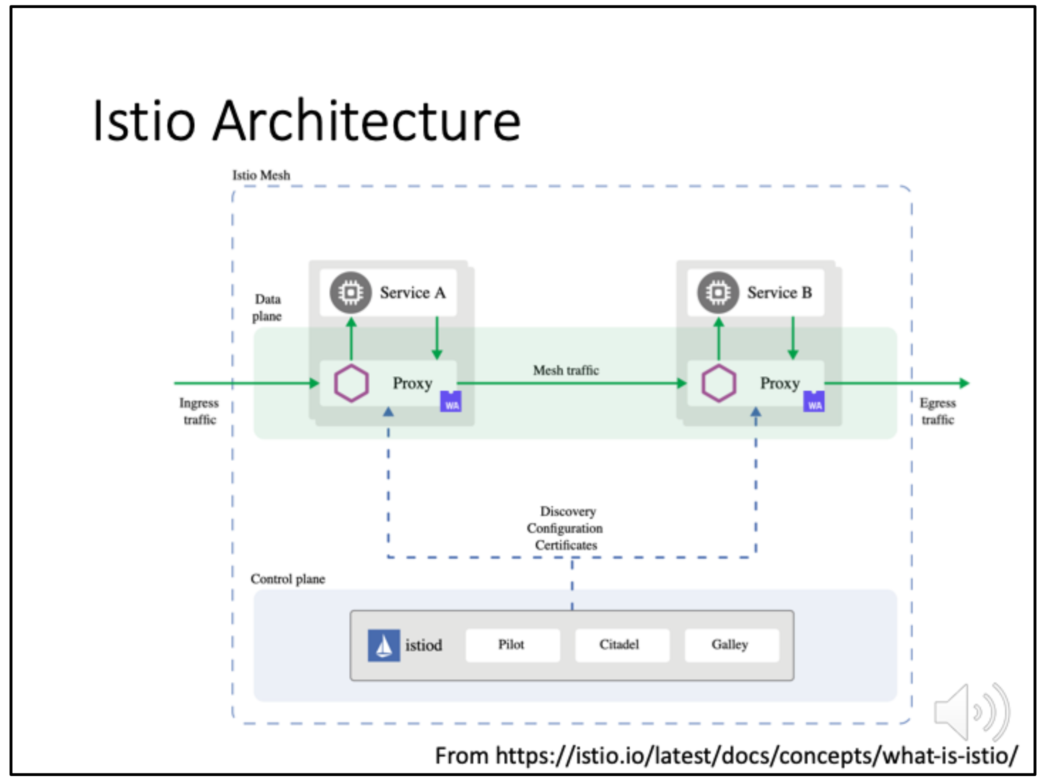


What will we be talking about today?

Istio, starting with 1.5, allows users to define custom code that runs when microservices
send or receive traffic.

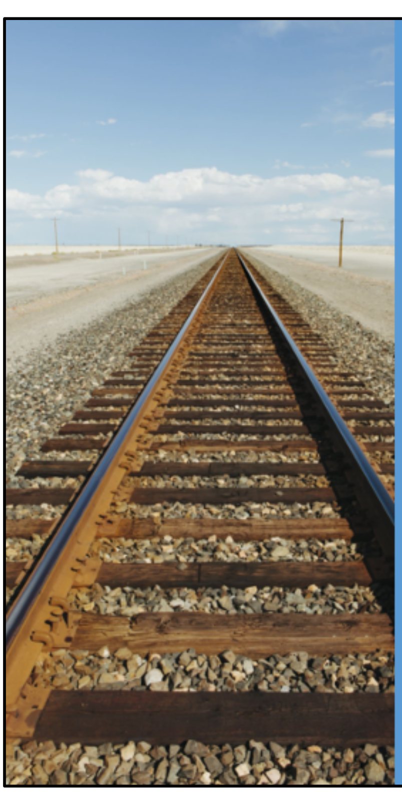
We will learn about the technology WebAssembly, or “Wasm”, behind this ability.

I will show you how to write and deploy Wasm extension plugins on your cluster using Istio,
and introduce you to a tool for creating and sharing your own WebAssembly Extensions.



To level-set everyone, here is the Istio architecture. Users write microservices. Istio attaches a proxy sidecar to them that manages and allows the microservices to participate in the mesh. Istio applies configuration and rules which are enforced by the sidecar.

Envoy is an edge proxy from Lyft. Envoy runs in a container in user pods. I will talk about extending Envoy with WebAssembly today. Everything we talk about today runs inside this component.



Why is WebAssembly for Proxies different and better?

- EnvoyFilter has existed since Istio 1.0
- In the past, you had to either create a custom build of Envoy with C++ extensions, or use an unsupported Lua
- **More languages, more power**

What is WebAssembly for Proxies, and why is it different and better than what we have had before?
After all, Istio has had EnvoyFilter since version 1.0.

In the past, you had to create your extensions in C++ and statically link them to Envoy. Users needing custom extensions couldn’t use
pre-build Envoy binaries from Lyft or the Istio project.

Although you could use Lua, it is several years old and unsupported.

WebAssembly gives us more languages and more power.

What is WebAssembly (Wasm)?

- Designed to execute within and integrate with the existing **web platforms**
- A portable, pre-compiled, cross-platform binary instruction set for a virtual machine that runs in the browser
- Not a new language; target for compilation of high-level languages

Why are we interested?

WebAssembly was designed to allow code to run inside web browsers in languages other than JavaScript. WebAssembly is a binary format. WebAssembly was designed to run in a web browser virtual machine.

A browser VM is not the same as an operating system VM. A browser VM is a sandbox similar to the Java Virtual Machine.

Instead of directly executing code like early versions of JavaScript, WebAssembly executes binary code. It isn't a new high-level language.

Today, we won't be talking about running WebAssembly on the browser. WebAssembly can now run server-side, on proxies.

Language support

- SDK and runtimes for **C++**
  - https://github.com/proxy-wasm/proxy-wasm-cpp-sdk
  - Mature
- SDK and runtime for **AssemblyScript**
  - Like TypeScript or JavaScript
  - https://github.com/solo-io/proxy-runtime
  - Somewhat mature
- **Rust**
  - https://github.com/proxy-wasm/proxy-wasm-rust-sdk
- If you are using another WASM language like **TinyGo** you are on your own
  - Application Binary Interface Spec
    - https://github.com/proxy-wasm/spec
- All Allow running Wasm in a VM (such as V8) within the Envoy proxy executable
  - A JavaScript Engine "VM", not an x86 virtual machine
- Simple single threaded model similar to Envoy

What features and languages will be available?

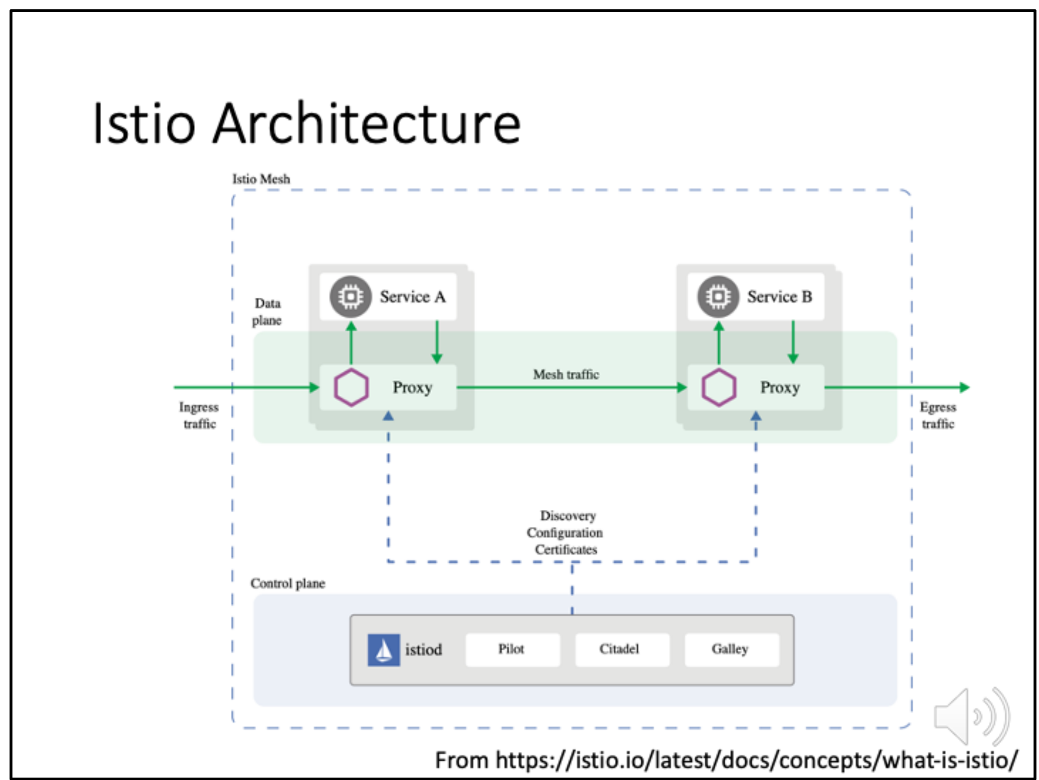
We have C++, AssemblyScript, and Rust (which I have not used).

I am most familiar with AssemblyScript. I will be show an AssemblyScript demo today. It's very much like JavaScript. It's NOT assembly language for WebAssembly. It is more like JavaScript for WebAssembly.

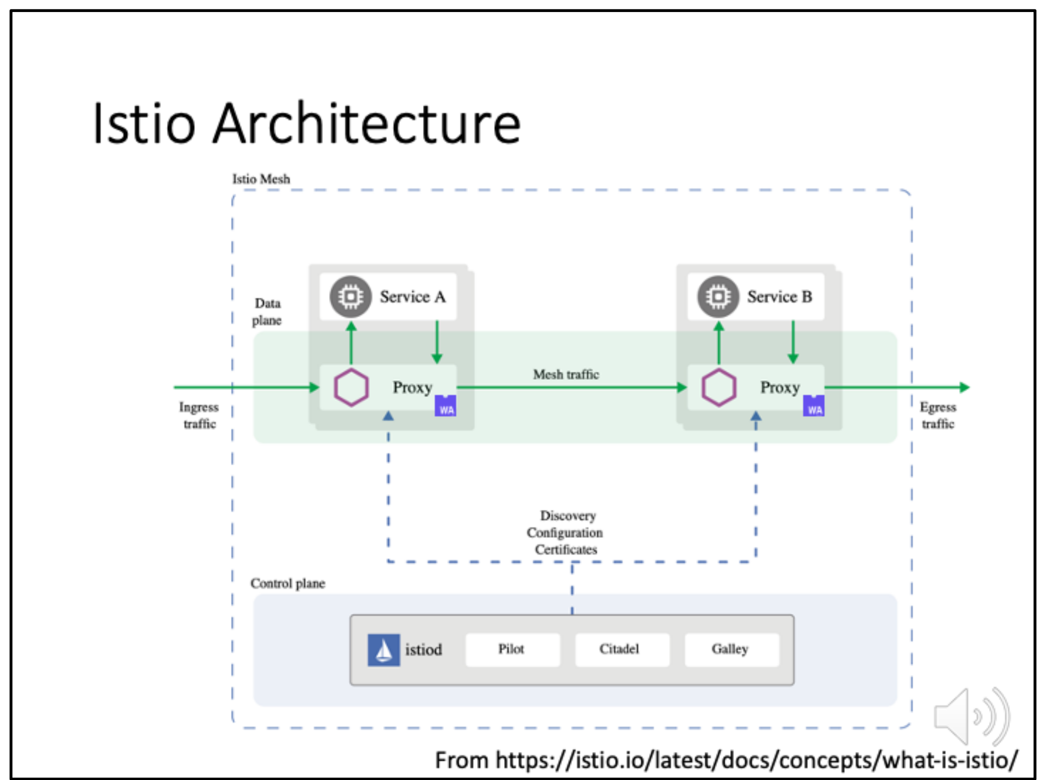
Other languages compile into WASM, such as a version of Go, but as of yet there is no Envoy SDK for Go. You are on your own if you want to use a language other than the ones here.

There is an Application Binary Interface Spec you would use to do that. We are not going to talk about that today.

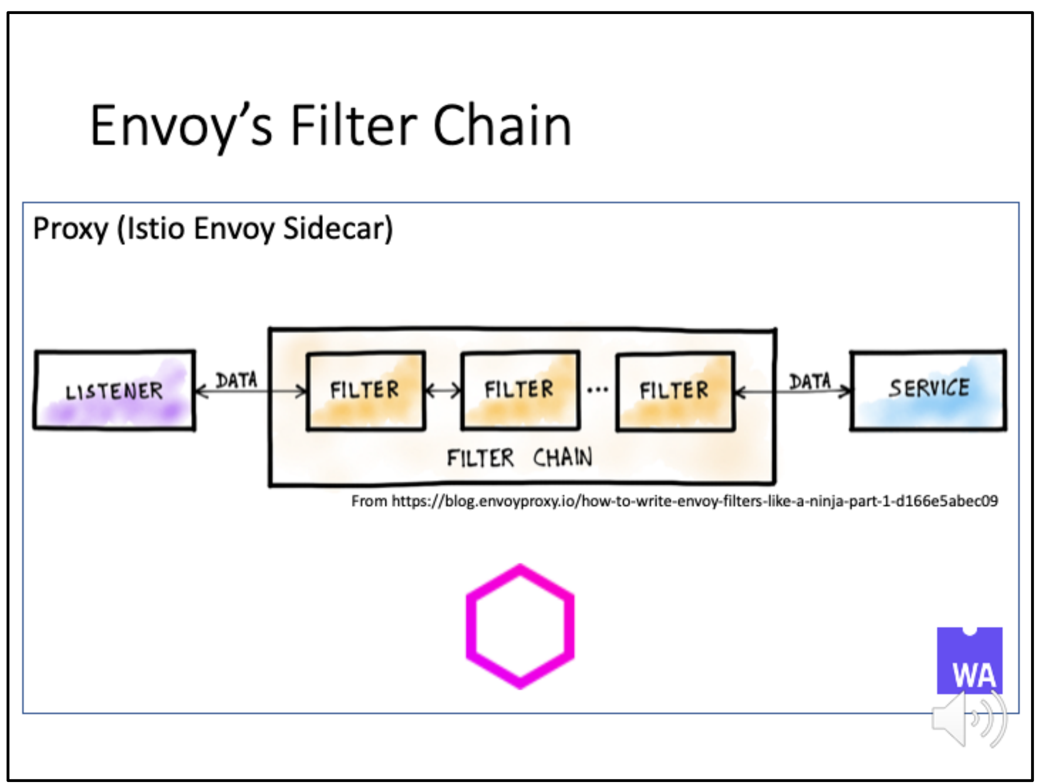
All of these languages only provide a simple single-threaded model when run on a proxy.
You will not have the full power of the language. You also have limited library support.



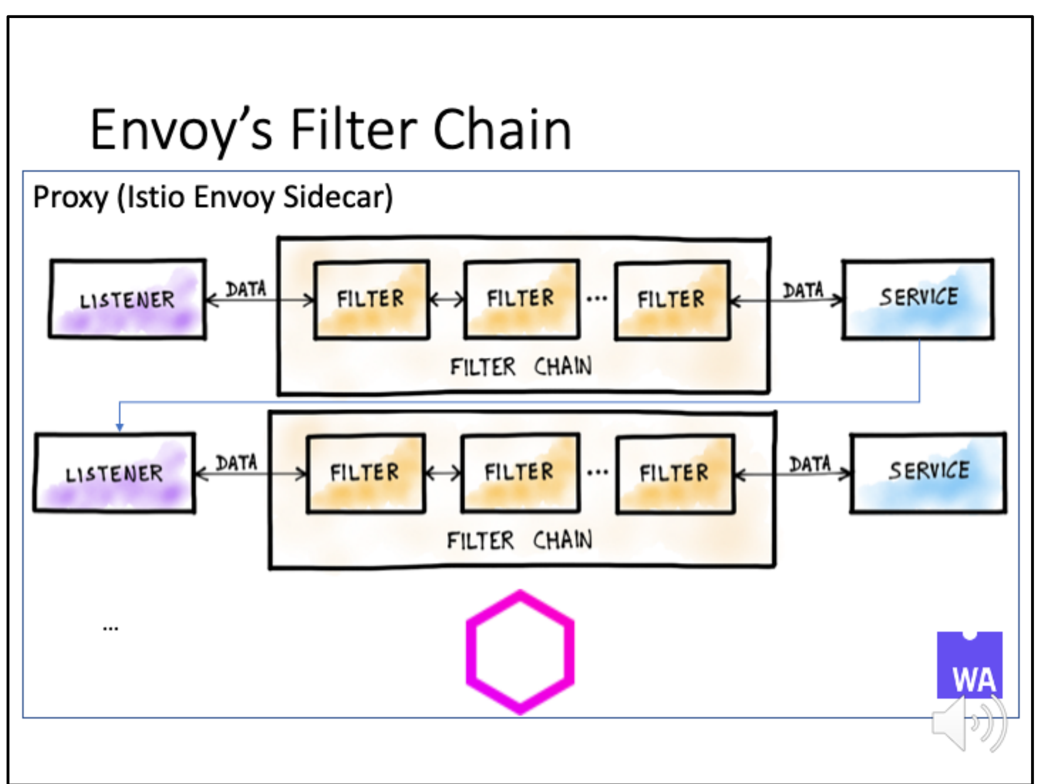
Envoy runs inside the box labeled “proxy”. I will talk about extending Envoy with WebAssembly today. Everything we talk about today runs inside this component.



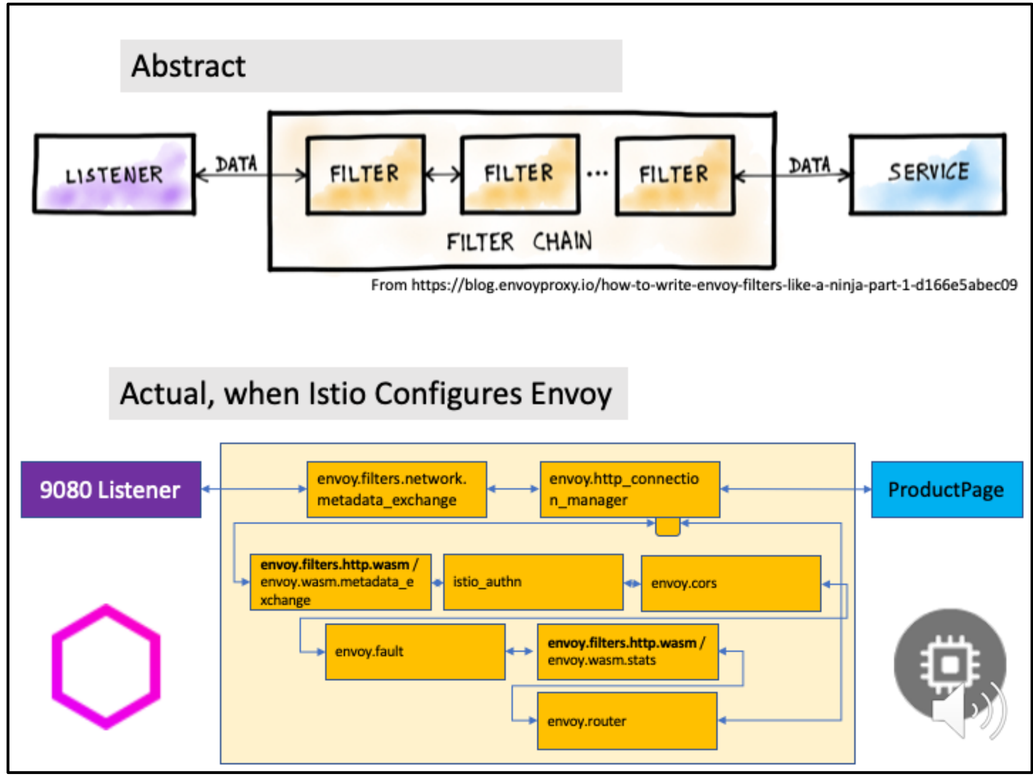
There is now a WebAssembly engine inside this proxy. Let's zoom into the proxy and see how it works.



The box here, this whole screen, represents a single sidecar from the previous slide. The internal architecture is that
there is a listener, and filters on inbound web traffic.



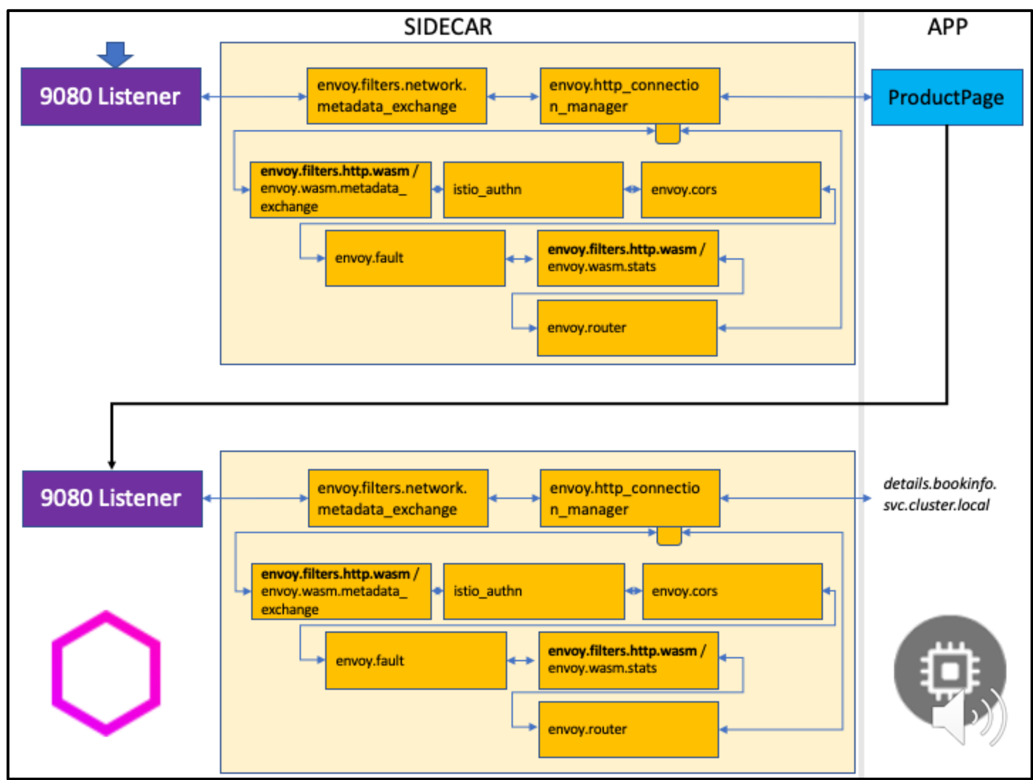
Listeners and filters also exist also before outbound services. More realistically, there will be dozens or hundreds of listeners, one from each inbound or outbound traffic route for your service.



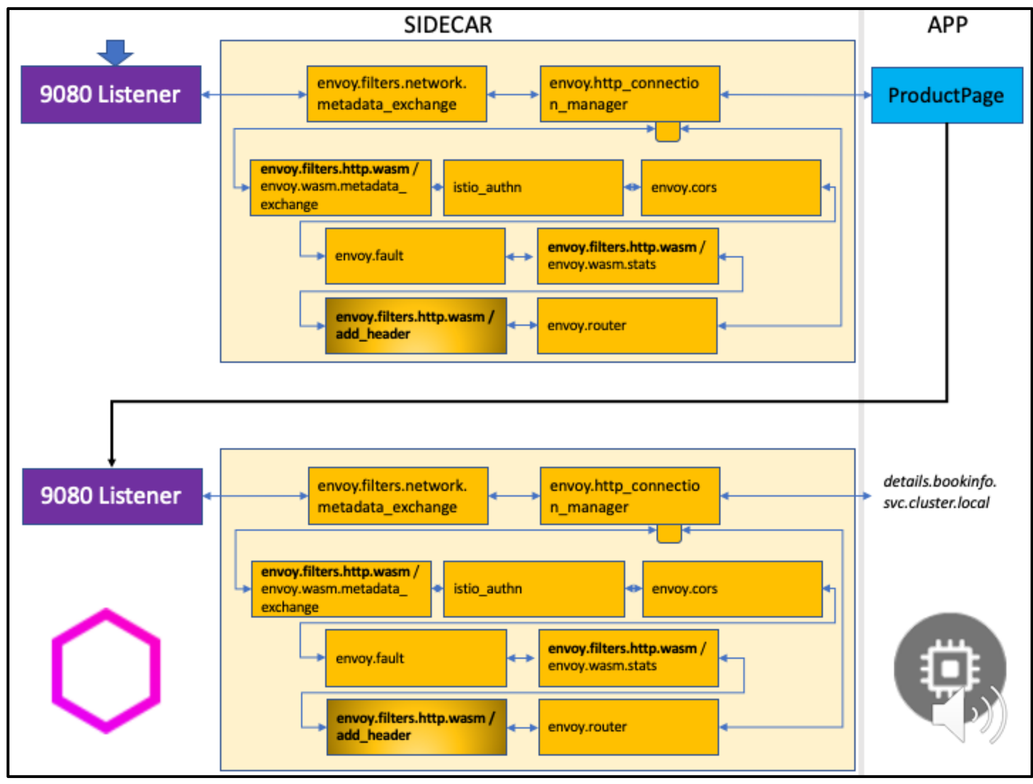
A more realistic picture shows that Envoy is listening on TCP port 9080.

When requests come in from the outside world, Envoy gives many filters a chance to block or modify traffic.

The first two filters are in generic filters. The six lower filters are HTTP specific. (For a TCP connection, you get something slightly different)
Envoy passes traffic through each filter in turn. Control flow can be blocked but usually reaches the end of the filter chain. Envoy then invokes the endpoint, in this case the ProductPage microservice app.

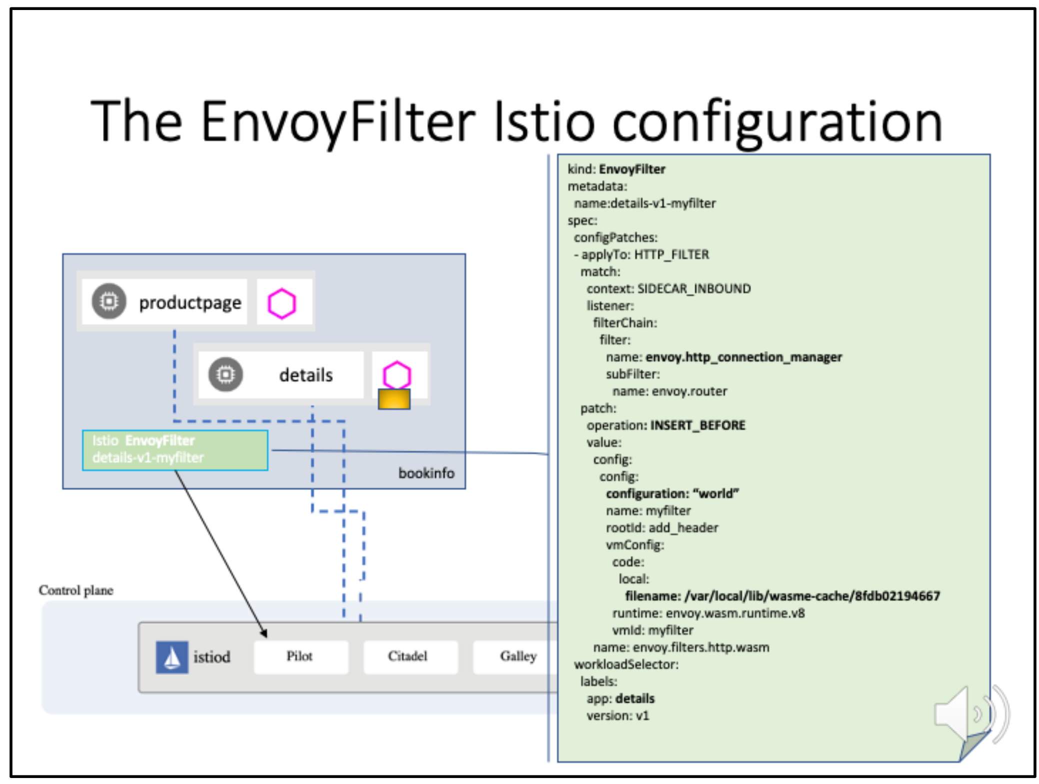


When ProductPage calls externally, the outbound side, it also through a set of filters. Configuring these filters gives Istio the ability to modify behavior without touching the Application's code at all.



This entire talk is going to be about adding a new piece onto this chain.
What I’m showing here is a very simple piece called “add header”. It is the piece used in the tutorial from Solo.io.

Let's learn how to add a single new filter to your sidecar within an Istio environment.



Let us recall how Envoy is configured.

In a static deployment of Envoy, without Istio, there would be an Envoy configuration text file. Developers modify that file to define behavior before building their Docker image.

Istio allows users to to reconfigure Envoy dynamically without touching the sidecar at all.

The sidecar receives pushes from the Istio control plane.

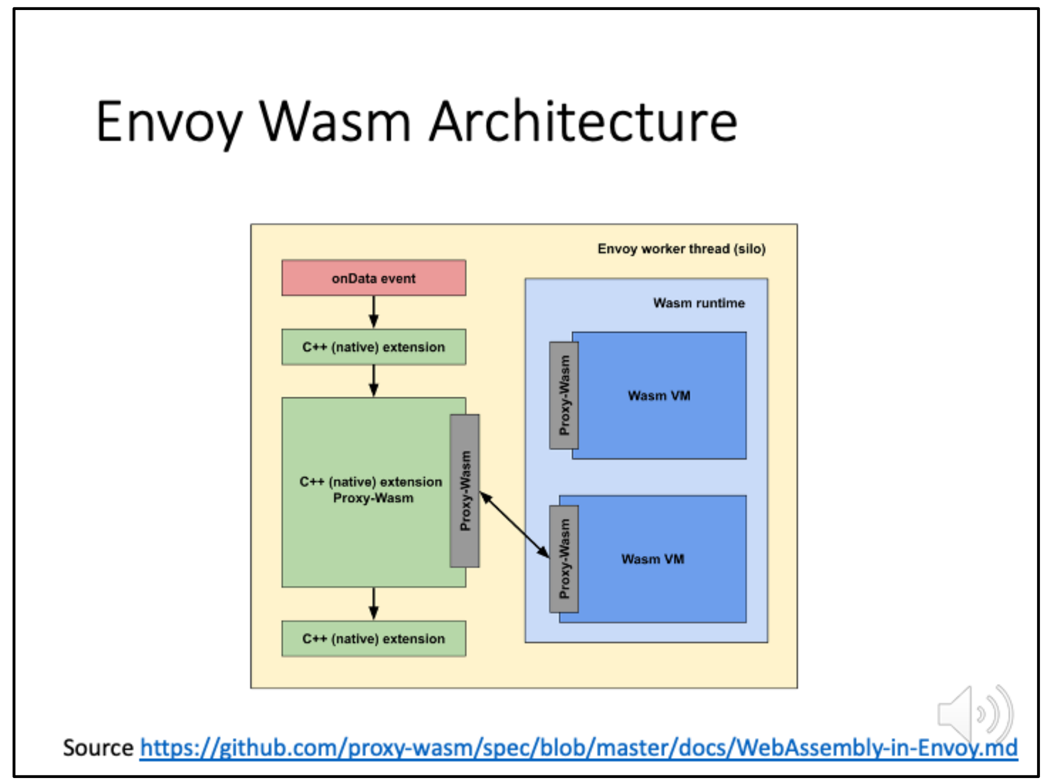
To add an Istio filter to pods, users add a custom resource called an EnvoyFilter to their cluster. When Pilot becomes aware of the new filter instance it pushes to all pods a new configuration.

The type of filter chain modification Istio does is controlled by EnvoyFilter resources like this one.

This example applies to the Details pod. It inserts this configuration — including *configuration* — to Details version 1.

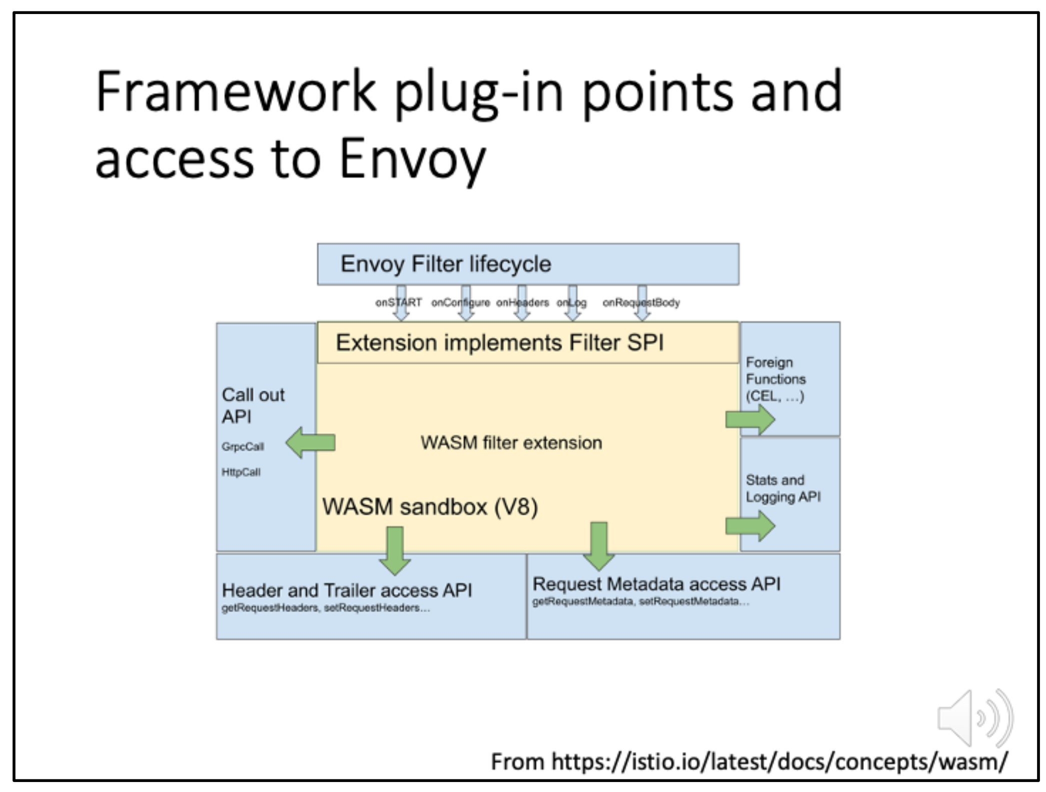
This EnvoyFilter Istio resource only defines attaching the filter to Envoy’s filter chain.

The compiled WASM binary code is declared to be in this location. Another mechanism, such as mounting storage, is needed to get the code to that location.



Once the code is present on Envoy and part of the filter chain, as new web requests arrive, that data goes through the regular C++ extension and eventually reach the green native WASM controller.

The extension will callout and execute, within this light blue WASM runtime sandbox, any WASM filters registered on the Envoy filter chain configuration.



Here is another view of how this works. Custom WebAssembly code is represented by this yellow box.
Envoy itself calls the code for lifecycle events like starting or configuring the plugin, or when headers or request bodies are to be processed.

Wasm code gets a chance to handle events, call functions in Envoy, and return a pass/fail to let Envoy know if it should continue processing the chain.

Suppose the code needs to alter the HTTP request. Custom code in the yellow has access to the headers and
can make changes to the request headers before the application sees it, Alternately, it may manipulate the response headers after the app container has set them.

The code could request an HTTP call to check with an authorization server. If the authorization server rejects, the WASM code would
return a value to Envoy to fail this call, and block the request from reaching subsequent filters; effectively blocking the traffic.

Wasme Commands

- **Open source user-facing tooling from Solo.io**
  - https://github.com/solo-io/wasme
- Create the directory with necessary files
  `wasme init`
- Build the Wasm binary
  `wasme build`
- Pull/push from/to the WebAssemblyHub
  `wasme push/pull`
- Deploy/undeploy the extension
  `wasme deploy/undeploy`

You could compile your C++ or AssemblyScript plugin yourself, put it into Kubernetes storage and mount that storage upon the istio-proxy container.

What I'll be showing on this talk is tooling from Solo.io, called *wasme.*
This tooling helps you do these activities in an easier way.

Wasme includes a command to create project boilerplate, setting up the things needed for a plugin either in C++ or AssemblyScript.
Wasme also sets up Envoy function library definitions.

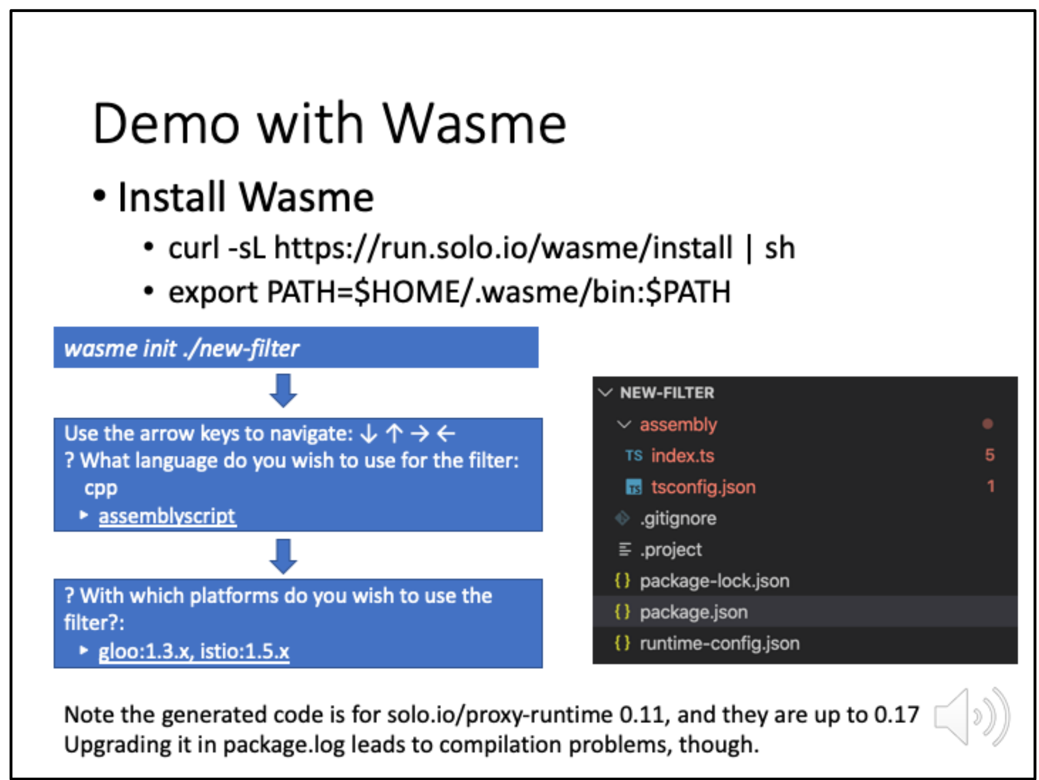
Wasme provides a *build* Command that executes the C++ or AssemblyScript compiler.

Wasme provides the ability to push and share your extension with others. Currently there is no way to push private extensions.

Wasme provides the ability to deploy the plugin to a cluster.

If you are only interested in using other people's plugins then you only need the last step.

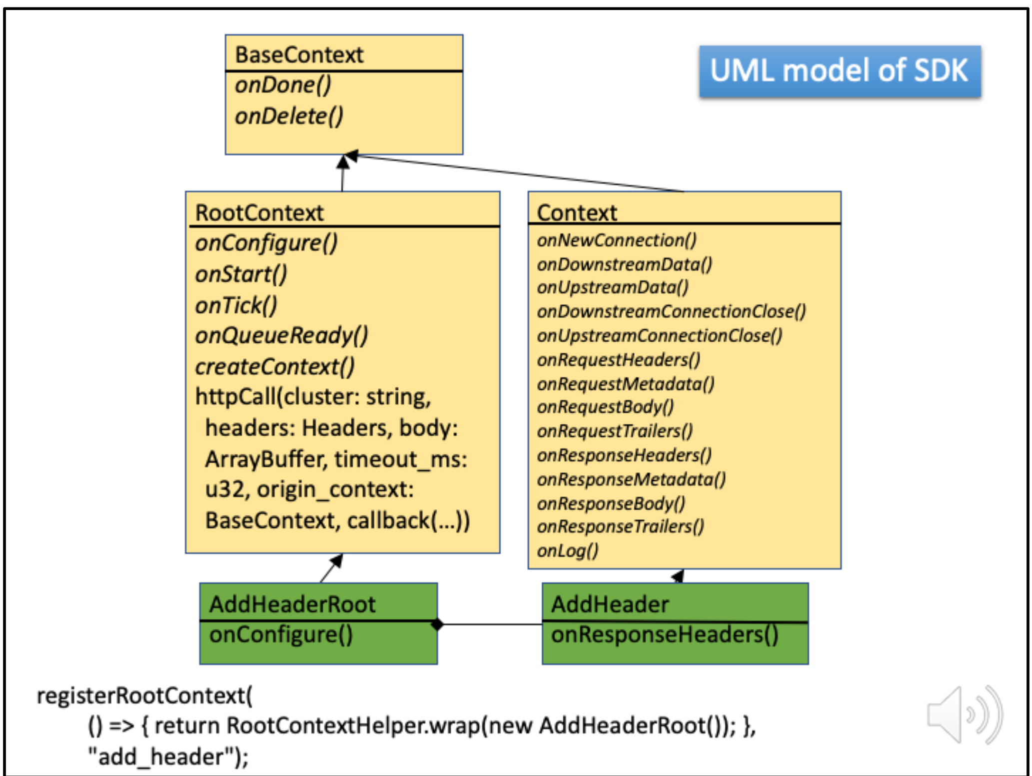
The first steps are about creating plugins. This is what I will show next.



In this demo, I will create a plugin called new-filter.

After installing wasme, I do *wasme init new-filter*
Two questions appear and I answer them interactively.

init creates the following files, including
- A typescript implementation of a filter
- The list of packages I need to run that filter.



Whether in C++ or AssemblyScript, the SDK gives developers an object-oriented view of an Envoy filter "context".

The hook points available to plugins are events like configure, start, timer-ticks header handling, request data handling, and processing an HTTP request flowing through the filters.

Your job is to subclass these and provide your own implementation of how to handle the events of related to the functionality your filter provides.

Plugin code must register statically by calling registerRootContext(). This call hooks your filter into the filter chain. Because your code extended RootContext or Context it is compatible with Envoy's expectations.

```
class AddHeader extends Context {
     root_context : AddHeaderRoot;
     constructor(root_context:AddHeaderRoot){
          super();
          this.root_context = root_context;
     }
     onResponseHeaders(a: u32): FilterHeadersStatusValues {
          const root_context = this.root_context;
          if (root_context.configuration == "") {
               stream_context.headers.response.add("hello", "(empty configuration)");
          } else {
               // root_context.configuration is loaded from the EnvoyFilter CR
               stream_context.headers.response.add("hello", root_context.configuration);
          }
          log(LogLevelValues.warn, "onResponseHeaders() context id: " +
                    this.context_id.toString());
          return FilterHeadersStatusValues.Continue;
     }
}
```

```
kind: EnvoyFilter
...
    config:
      configuration: "world"
```

wasme build assemblyscript -t webassemblyhub.io/$YOUR_USERNAME/add-header:v0.1 .
*wasme push webassemblyhub.io/$YOUR_USERNAME/add-header:$WTAG && *
*wasme deploy istio webassemblyhub.io/$YOUR_USERNAME/add-header:$WTAG *
--id=myfilter --namespace bookinfo --config 'world'

Here is example code for an HTTP filter. This code was written by *wasme init*, but I added logging.

The developer defines onResponseHeaders() ensuring this Wasm plugin will be called with HTTP response headers. The code adds a header. The header name, “hello”, will be compiled into the binary code. The header value, “world”, will come from an Istio EnvoyFilter resource.

The developer performs *wasme build* and an AssemblyScript project is compiled into a binary.
The developer tags the build as *webassemblyhub.io/snible/add-header*:v0.1

The developer next pushes it to the WebAssembly hub. This step pushes compiled code from a laptop to solo.io’s hub site. We will look at that process later.

Next, a Kubernetes user deploys the filter upon a cluster, customizing it with the value “world” we will inject into the headers.

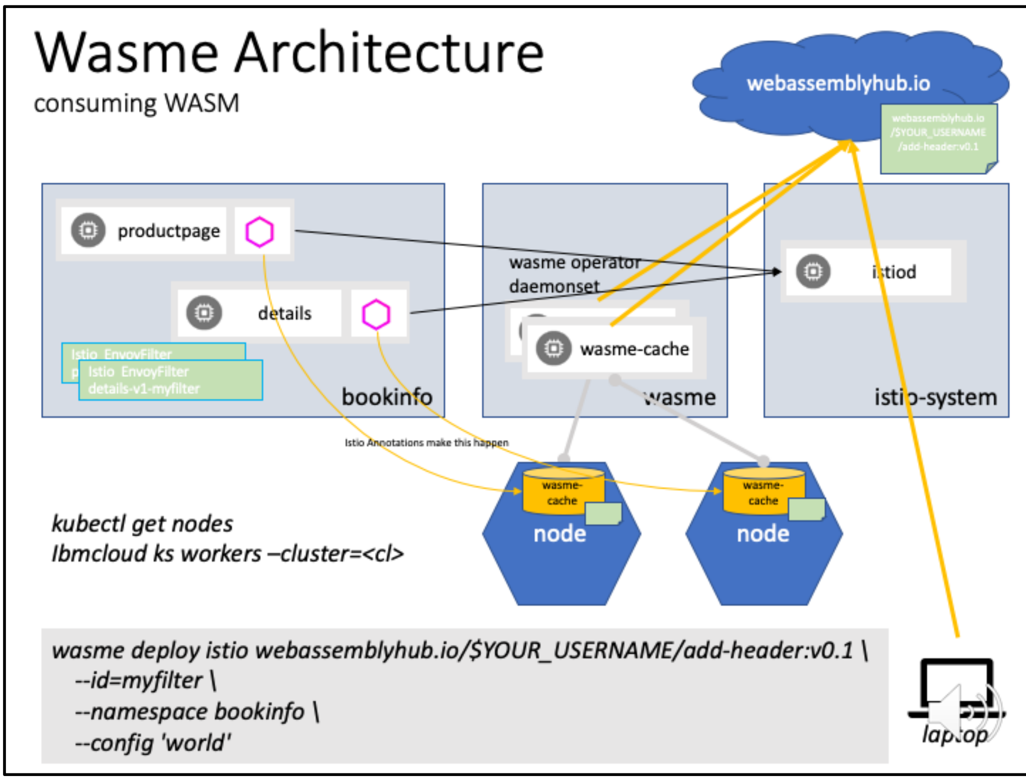
```
~/src/wasme/new-filter$ kubectl exec -ti -n bookinfo deploy/productpage-v1 -c
istio-proxy -- curl -v http://details.bookinfo:9080/details/123

*   Trying 172.21.61.226...
* TCP_NODELAY set
* Connected to details.bookinfo (172.21.61.226) port 9080 (#0)
> GET /details/123 HTTP/1.1
> Host: details.bookinfo:9080
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: application/json
< server: istio-envoy
< date: Mon, 15 Jun 2020 19:04:29 GMT
< content-length: 180
< x-envoy-upstream-service-time: 1
< hello: world
< x-envoy-decorator-operation: details.bookinfo.svc.cluster.local:9080/*
<
* Connection #0 to host details.bookinfo left intact
{"id":123,"author":"William
Shakespeare","year":1595,"type":"paperback","pages":200,"publisher":"PublisherA",
"language":"English","ISBN-10":"1234567890","ISBN-13":"123-1234567890"}
~/src/wasme/new-filter$
```

```
kubectl exec -ti -n bookinfo deploy/productpage-v1 -c istio-proxy \
  -- curl -v http://details.bookinfo:9080/details/123
```

To interactively test it, I use *kubectl exec* to force productpage do a *curl* on details and we can see the new header “hello: world” is present.

In the real world changes wouldn’t be tested this way, but this is good enough to learn and develop small Wasm extensions.



This diagram shows the process code of compiling code and the flow from a development machine to an istio-proxy sidecar.

After compiling the code, developers do a *wasme push* to copy the compiled code to a cloud server.

Next operators perform a *deploy*. The first time operators deploy, *wasme* creates a new namespace in the cluster, called also *wasme,* and creates a Kubernetes operator there, implemented as a daemonset. (A daemonset is a Kubernetes concept of a deployment with one pod per node.)

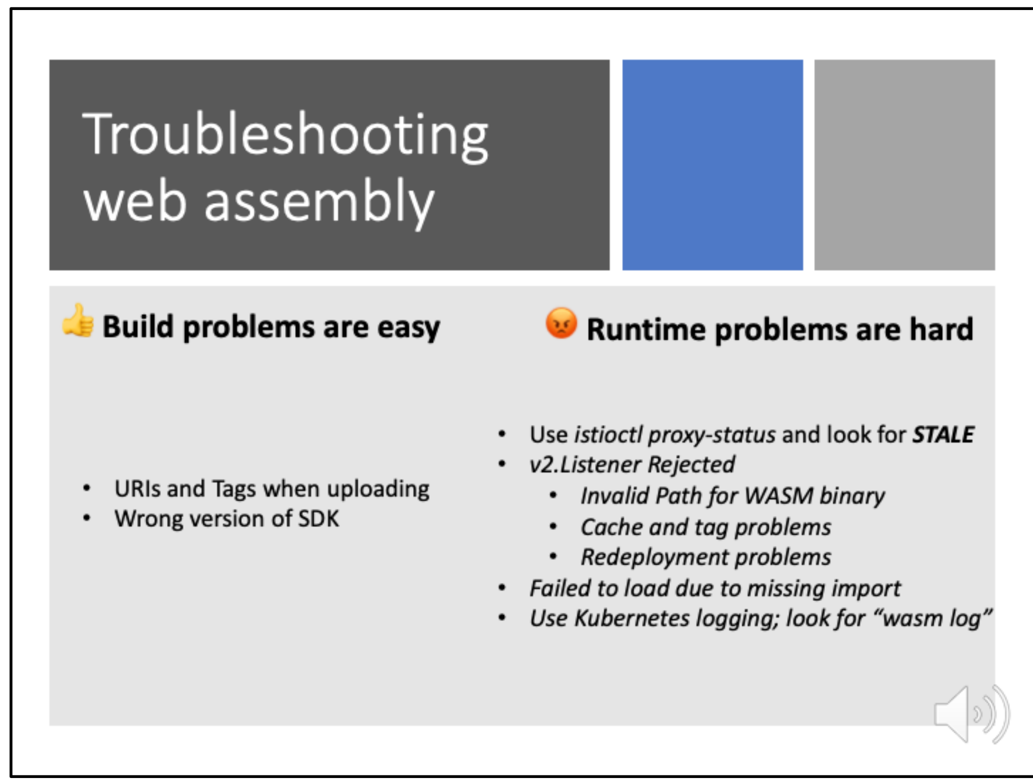
For example, on my cluster I have two nodes. Those are the nodes seen with *kubectl get nodes* or
ibmcloud kubernetes-service workers

This daemonset causes there to be a local file, on every worker node, that contains a cached copy
of what has been deployed on the WebAssembly hub.

wasme deploy first pulls the code and puts it in this cache, where it will be available to every pod running upon that node.

Next *wasme* modifies Deployment pod template by adding annotations so the sidecar sees that node cache as a directory on the sidecar.

Wasme creates EnvoyFilters to apply the file in the cache to a sidecar’s filter chain.



Some common problems you may encounter when developing plugins.

When I started building, I had many problems with the webassembly URI and tags.

Later, when I tried to use new Envoy Wasm features, I ran into problems with Assemblyscript SDK versions.

Problems with building are easy to detect.

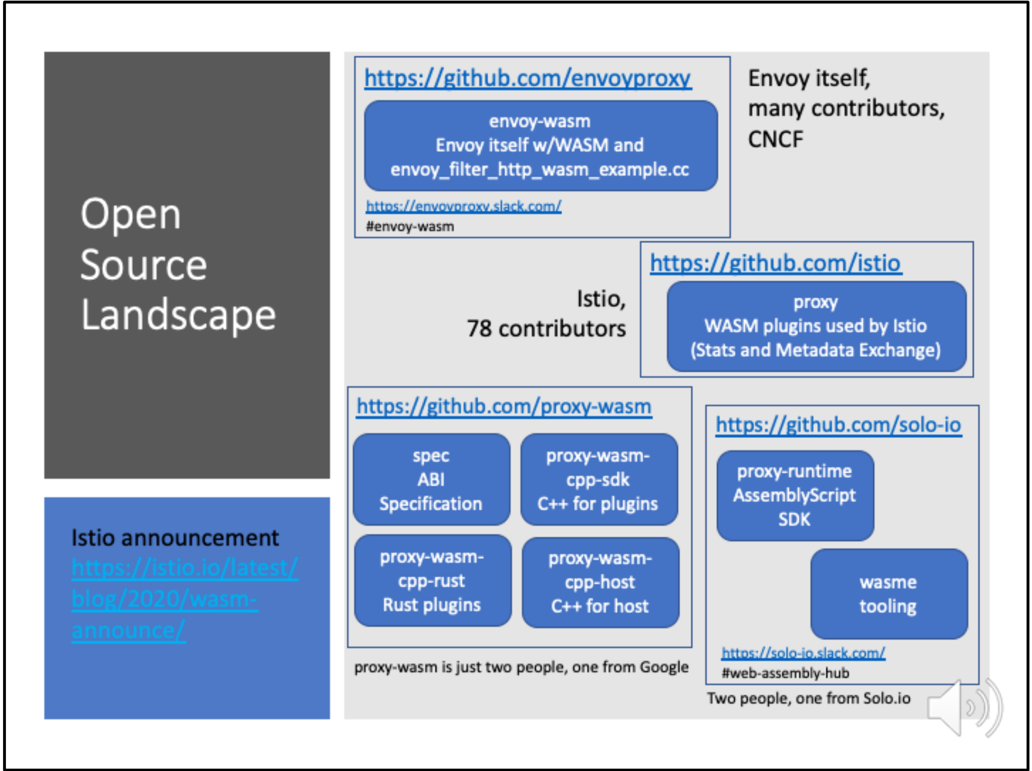
If you have problems with your compiled plugins not working these can be easy to miss.

Use *istioctl proxy-status* to make sure the sidecars aren't rejecting the configuration created by Wasme.

Common problems leading to stale proxies include redeploying Deployments. Wasme modifies the Deployment, but not the Istio injector, and the annotations that mount the cache can get lost.

Also, references to functions that are not defined by the Envoy version you are using can also cause rejects.

If the proxy is in sync, but isn't working, add logging to your code. Search Envoy's logs for lines with "wasm log" to see the logs produced by Wasm extensions.



Wasm support shipped with Istio 1.5. You can read the announcement here.

To learn more about WebAssembly, there is a lot of material on the Internet covering WebAssembly for browsers.

To learn about Wawm for Proxies and get more involved there are some fundamental Github projects to examine.

The biggest piece comes from Envoy itself - a special version called envoy-wasm which includes a sample C++ wasm filter.

The istio-proxy project includes code for Istio's stats and metadata exchange filters, the pieces of Istio that control telemetry.

The SDKs for C++ and Rust come from the proxy-wasm project, and also the specification for how plugins interact with Envoy.

The AssemblyScript SDK and *wasme* tooling that I showed come from Solo.io.

There are also public Slack channels for Envoy-Wasm and the WebAssemblyHub.

Any questions?

- Any questions?

Any questions?