

info Creating precompiles requires writing lower level code than the contracts and transactions we've been working with so far. You will need a local clone of `suave-eth` to follow this tutorial, and we'll be making changes to the core code itself. SUAVE uses [custom precompiles](#) to extend the EVM with specific MEV functions. Unless you have a very specific use case, building a SUAPP should not require writing precompiles.

If you do want to create your own precompile, please [consult our governance process](#) along with this tutorial.

We specify the inputs and outputs for each precompile in `inyaml`. This automatically generates two bindings, one in Solidity (the client) and another one in Go (the server).

The bindings handle encoding/decoding and error management, providing a standard format for both runtimes to communicate with each other. This removes all the nitty-gritty work required such that you can focus on creating the precompiles you need without getting caught up in the implementation complexities.

types : -

```
name : BidId type : bytes16 structs : -
```

name : Bid fields : -

```
name : id type : BidId -
```

name : decryptionCondition type : uint64

functions :-

```
name : confidentialInputs address :
```

[illegible]

true fields : -

```
name : output1 type : bytes -
```

```
name : newBid address :
```

```
'0x0000000000000000000000000000000000042030000' input : -
```

name : decryptionCondition type : uint64 -

```
name : allowedPeekers type : address [ ] -
```

```
name : bidType type : string output : fields : -
```

name : bid type : Bid There are three top-level objects: types ,structs and functions . In this guide, we will focus on the functions , as adding a new precompile will most often entail writing a new function.

If you can specify the function's name, the address its logic is deployed at on SUAVE, and what form you expect the inputs and output to take, then our codegen tool will automatically generate both the Solidity and Go bindings required to make your precompile work.

The fields you can include when adding a new function to the yamlspecification are:

- name
- : Name of the precompile.
- address
- : Address of the precompile.
- input

- Input and output types can be a basic Solidity type (address), a composite type (address[]), or a reference to any of the custom types and structs (i.e. Struct, Struct[]). It must be written in the same format as it would be in Solidity.

Now we can write a custom SUAVE precompile to perform the "add" operation in order to illustrate how to add your own. This is a two step process:

- You can edit the [yaml specification here](#) . We'll add a new entry in the functions section:

name : output1 type : uint64 Then, run our code generator:

func

```
( b * suaveRuntime )
```

```
Add ( a uint64 , b uint64 )
```

```
( uint64 ,
```

```
error )
```

```
{ return a + b ,
```

```
nil }
```

You can find a worked example of how to add a more complicated precompile [in this PR](#) . [Edit this page](#) [Previous](#) [Confidential Compute Requests](#) [Next](#) [Tools](#)