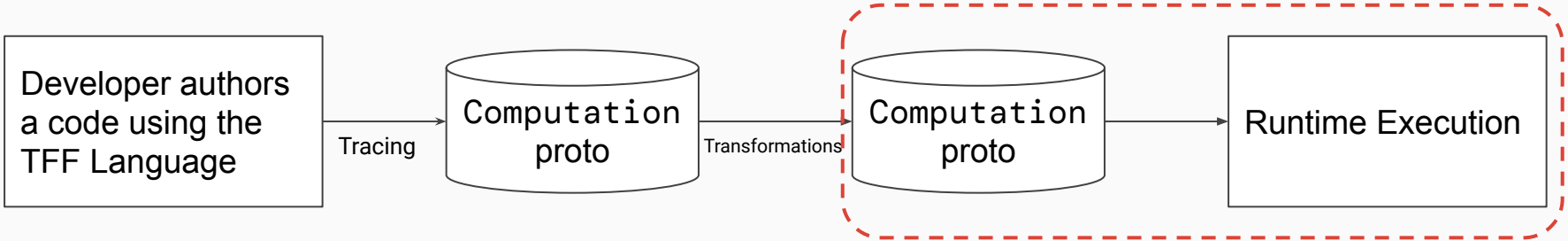
# TFF 102: Executors, Executor Stacks, and Execution Contexts

Last updated: 2022-07-06

zachgarrett@google.com, krush@google.com

# TFF introduction series

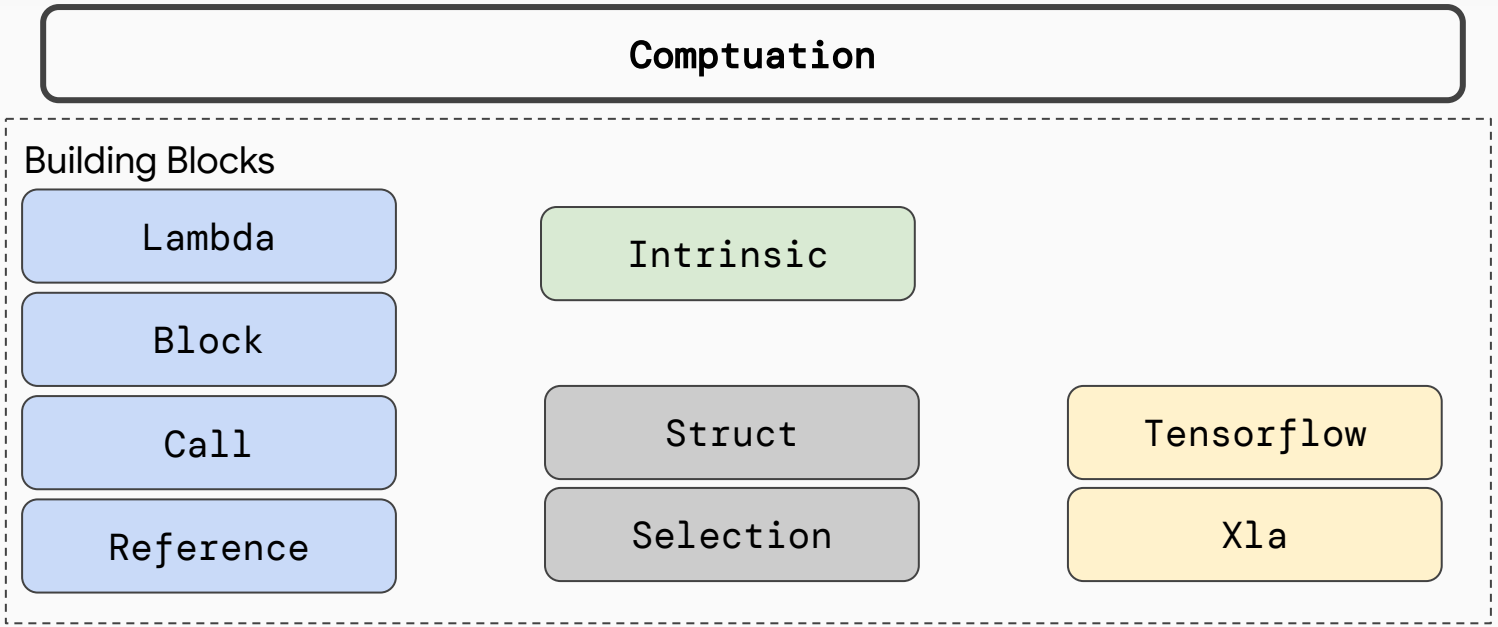
- TFF 101: Lingua Federated, Training, and ASTs
- **[this talk]** TFF 102: Executors, Executor Stacks, and Execution Contexts
- TFF 103: Transformations, Compiler Pipelines, and *Forms*




Life of a TFF computation

# Review: The `Computation` proto (TFF's AST)

https://github.com/tensorflow/federated/blob/main/tensorflow_federated/proto/v0/computation.proto



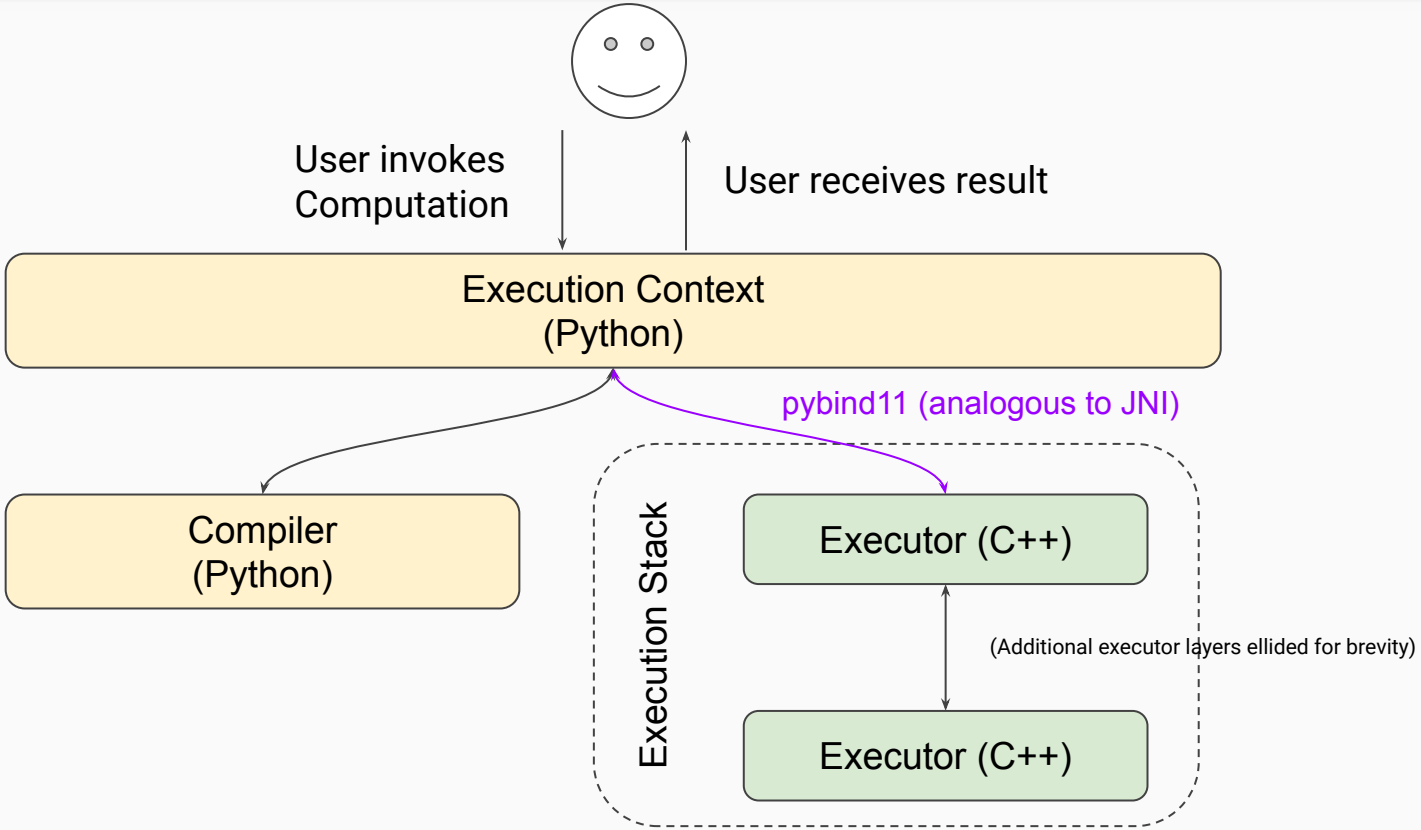
# Review: TFF Design Principles

1. Design composable pieces that can be built-up into larger structures that perform more powerful operations.
    a. Expressible interfaces, limited implementations: prefer that a single object or function does a single thing.
2. Flexible, or even better functional, parameterizations.
    a. Extend functionality via parameterizations.

# Today's question

What happens when invoking a `tff.Computation`?

# What happens when invoking a `tff.Computation`?




The 3,000 meter view of the execution of a `tff.Computation`

# Terminology we'll cover today

**Execution Context**

The integration point between an executor stack and callable TFF computations. Responsible for managing the executor stack, ensuring it is correctly configured to execute the incoming computation (e.g. compiles computations if necessary).

**Executor Stack**

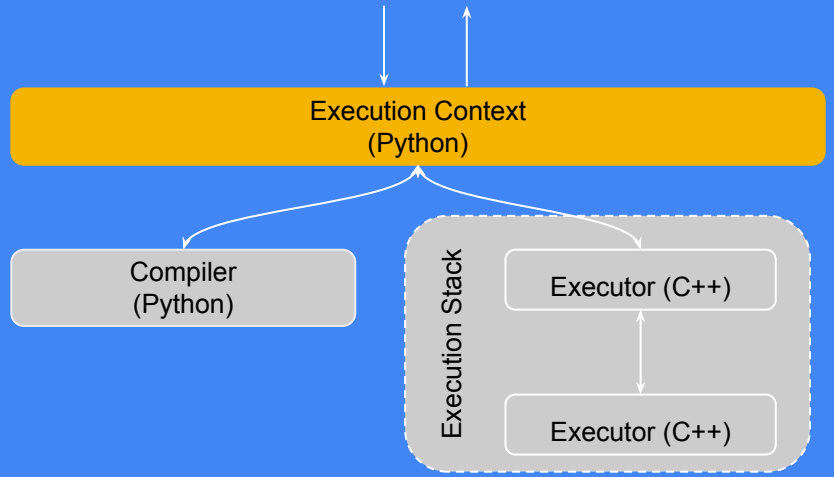
A composition of executors that provides the ability to execute different computations. The top of the stack evaluates the parts of the computation it understands, and delegates the remaining parts of the computation to its children.

**Executor**

Either the abstract API or a concrete implementation of an executor.

While possible to create a monolithic executor that "does it all", the general design philosophy is to focus on execution of one particular part of the language.

# Execution Context



# What is an execution context?

Manages the executor that a computation is executed in, as well as invocations of computations. **Provides guarantees to the user experience.**

Implementation of the abstract base class `context_base.Context` that has one method: `invoke`.

Again using Python `__dunder__` magic (remember **TFF 101**) the following:

```
foo(x)  # foo is a `tff.Computation`
```

will end up calling:

```
execution_context.invoke(foo, x)
```

# What is the job of execution context?

Mediate between the user (who just called a TFF Computation) and the TFF runtime.

- The context is the thing responsible for bridging the callable interface of a TFF computation and the `Executor` interface that the TFF runtime implements.
  - These two interfaces may have different desired semantics (e.g., in what situation they raise errors); the context implements the one in terms of the other.
- We prefer to do **less** in the context if possible. Since it's a layer on top of `Executor`, things implemented here are not composable with the rest of the runtime.

# What do I do with an execution Context?

A `Context` is one possible extension point!

If you want ‘new behavior’ when your computation is invoked, you *can* implement your own context.

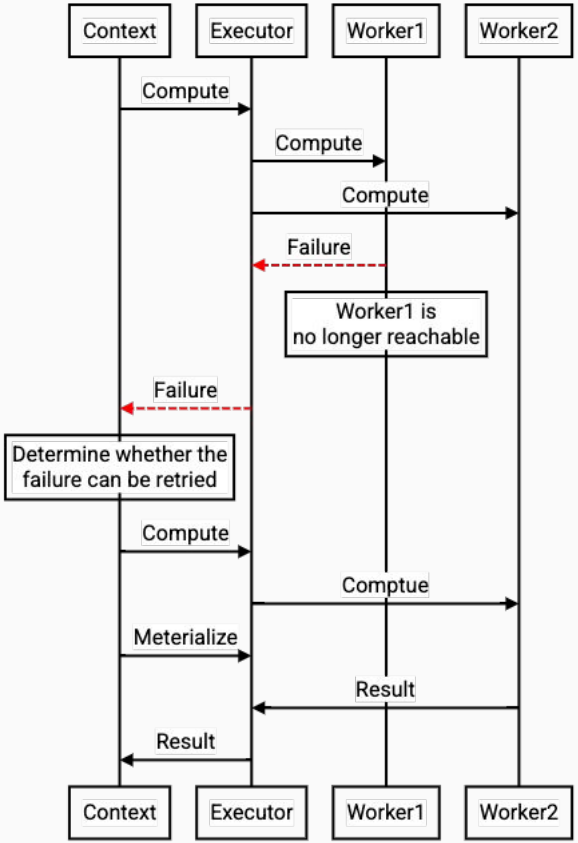
```python
import asyncio
import tensorflow_federated as tff

class MyAsyncContext(tff.framework.Context):
  """A minimal execution context for asynchronous calls into TFF.

  This context implements no error handling or retries, and is not configurable
  based on cardinalities, etc. of the argument.
  """

  def __init__(self, executor: tff.framework.Executor, materialize_return_value: bool = True):
    self._executor = executor
    self._materialize_return_value = materialize_return_value

  async def invoke(self, comp, arg):
    comp_bb, _ = tff.framework.replace_intrinsics_with_bodies(comp.to_building_block())
    embedded_comp = await self._executor.create_value(comp_bb.proto)
    if arg is not None:
      embedded_arg = await self._executor.create_value(arg, comp.type_signature.parameter)
      call = await self._executor.create_call(embedded_comp, embedded_arg)
    else:
      call = await self._executor.create_call(embedded_comp, None)
    if self._materialize_return_value:
      return await call.compute()
    else:
      # Returning a handle like this is nice if you can do it. TFF currently
      # always calls materialize before returning to avoid trying to preserve
      # the validity of 'call', which may represent a distributed value. There
      # is a future in which this is possible to support--we'll see if it's the
      # universe we're living in.
      return call
```

# What does a `Context` do by default?

The context is usually parameterized by several functional components, allowing the context to compute information necessary to 'orchestrate' the invocation. Normally, this includes:

- Constructing an instance of `Executor` which can execute the given `computation`, using all available resources.
- Normalizing (or 'compiling', more in **TFF 103**) the computation to a form the `Executor` expects.
- Calling the necessary methods on the `Executor` to compute the result of `comp(arg)`.
- Possibly handling exceptions raised by the `Executor`, e.g. retrying if a worker fails.
- Repackaging results into a form the user expects

# Example: worker failure



# How can I extend it?

As noted, `Context` can be subclassed. But default implementations additionally expose at least three major extension points today:

- `ExecutorFactory`
  - A ~function which accepts **cardinalities** (more on cardinalities in a few slide) and returns an `Executor` on which the context will make invocations. Factories can manage `Executor` caches, be configured to inspect the runtime environment and, e.g., ensure work is partitioned evenly across workers, etc.

# How can I extend it (con't)?

- **Compiler**, a function which accepts a `tff.Computation` and returns one (or more) `tff.Computations`, as the context expects. Compilers may, for example, lower intrinsics to a supported set, compile local computations to TF or XLA, rearrange the computation, split it apart, etc.
- `Context` is **composable** (recall *design principles of TFF* from TFF 101); it can be passed to another context, which performs some 'higher-level' orchestration.

# What information does the context need to supply?

**Cardinalities:** TFF computations are only fully-specified given some specification of a cohort, either implicitly or explicitly

```
@tff.federated_computation
def count_clients():
  return tff.federated_sum(
    tff.federated_value(1, tff.CLIENTS))
```

If you call this, what should you get back?

# Example: Cardinalities

Cardinalities information is not always necessary to specify.

- At times it can be inferred from arguments.
- At other times it must be implicit in the runtime's configuration.

A concrete instance of `Executor` is *always assumed to represent a single set of cardinalities*.

- There are real subtleties here in terms of the relationship between cohort and cardinalities. TFF does not always assume an `Executor` instance to represent a fixed **cohort** (though it can); but an instance of `Executor` will always represent a fixed 'set' of cardinalities.

# Composing contexts: leveraging mergeability*

- If a computation contains a single logical aggregation, it can be run in ‘subrounds’, leveraging the capacity to re-run the `merge` step of `tff.federated_aggregate`.
- Since retries are today handled at the `ExecutionContext` level, a context which accepts `ExecutionContexts` and orchestrates them in executing subrounds may be significantly more robust to worker failures than default TFF execution.
  - Leveraging this compositionality provides a simple way to scale TFF without designing new components.

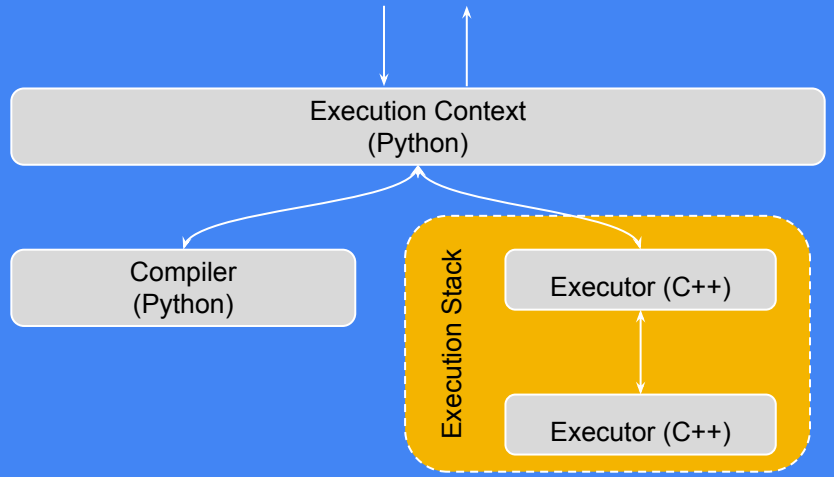
* TFF prefers to move this kind of functionality inside the Executor interface ‘eventually’

# Composing contexts: leveraging mergeability

```
async def invoke(self, comp, args):
  up_to_merge, merge, after_merge = self._compile(comp)
  # Run the subrounds in parallel. Each of the invocations
  # independently retries if one of their workers fails.
  subround_results = await asyncio.gather(
    x.invoke(up_to_merge, y) for x, y in zip(self._contexts, args))
  # Once we get here, every context has completed their work.
  merged = subround_results[0]
  for to_merge in subround_results[1:]:
    merged = await self._contexts[0].invoke(
        merge, (merged, to_merge))
  # Technically we need to run in every context then repackage,
  # but we suppress this for now.
  return await self._contexts[0].invoke(after_merge, merged)
```

With a little extra massaging, a ‘context which orchestrates contexts’ like the above has allowed TFF’s runtime to reliably handle 10K+-client rounds for these ‘mergeable’ computations. Arbitrary scale, at the expense of speed, can be had by sequentializing and checkpointing (not default today).

# Executor Stacks

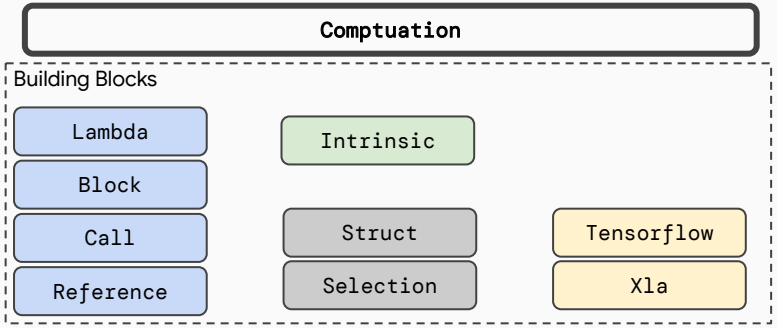


# TFF Design Principle: Composable components

An **Executor Stack** is a *composition* of executors.

Each **Executor** handles a specific part of execution, while **Execution Stacks** can be built to handle different shapes of computation/capabilities.

In general an Executor will handle some building blocks, while delegating other building blocks to other Executors (more on this later).




**Executor Implementations today**

- TensorFlowExecutor (.h)
- FederatingExecutor (.h)
- ComposingExecutor (.h)
- ReferenceResolvingExecutor (.h)
- SequenceExecutor (.h)
- DataExecutor (.h)
- RemoteExecutor (.h)

In the later section on ***Executors*** we will see how the executor interface works to communicate between Executors.

# What is the *minimal* execution stack?

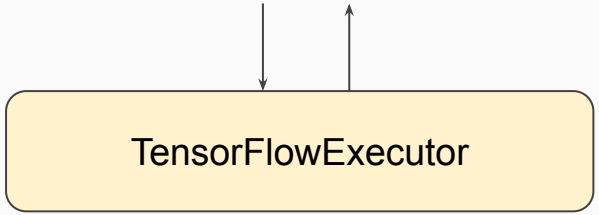
```
@tff.tf_computation(tf.int32)
def add_one(x):
  return x + 1
```

AST:

`comp#a0145b1c`

Computation proto:

`tensorflow { … }`




This computation only has a TensorFlow block, nothing else. It can be handled purely by a **TensorFlowExecutor**.

# What about for federated computations?

```
@tff.federated_computation(
  tf.int32)
def add_one(x):
  return x + 1
```

TFF AST:

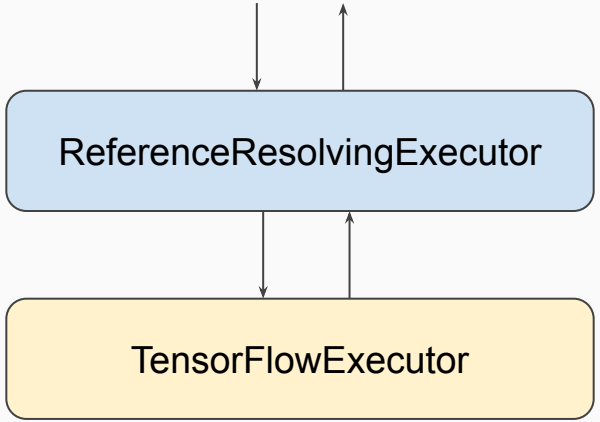
```
(add_one_arg -> (let
  fc_add_one_symbol_0=comp#5abc2acf(),
  fc_add_one_symbol_1=generic_plus(<
    add_one_arg,
    fc_add_one_symbol_0
  >)
 in fc_add_one_symbol_1))
```

Takeaway: addition is hard, leave it to TensorFlow.

Computation proto:

```
lambda {
  parameter_name: "add_one_arg"
  result {
    block {
      local {
        name: "fc_add_one_symbol_0"
        value {
          call {
              tensorflow { … }
          }
        }
      }
      local {
        name: "fc_add_one_symbol_1"
        value {
          call {
            intrinsic { … }
          }
        }
      }
      result {
        reference {
          name: "fc_add_one_symbol_1"
        }
      }
    }
  }
}
```

# Federated Contexts require at least resolving reference



# But the previous example didn't do anything "federated"

```
@tff.federated_computation(
  tff.types.at_server(tf.int32))
def multiply_by_num_clients(x):
  # Sum the incoming number across all
  # clients, effectively multiplying by
  # the number of clients.
  return tff.federated_sum(
    tff.federated_broadcast(x))
```

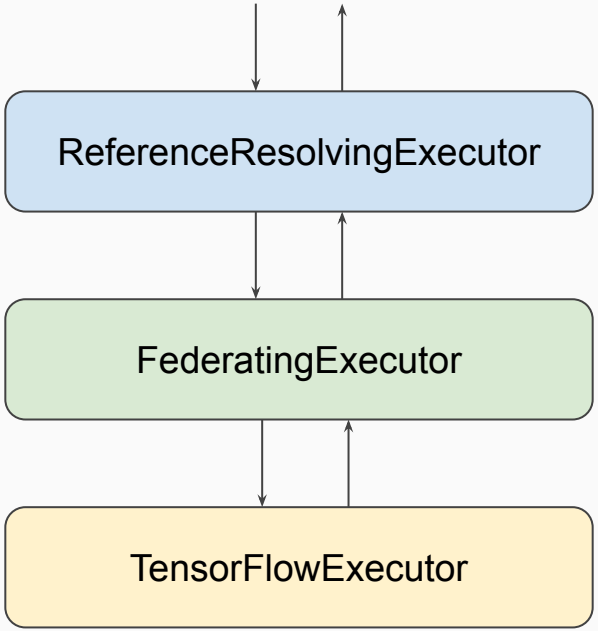
`Computation` proto:

Too big to fit in these slides.

TFF AST:

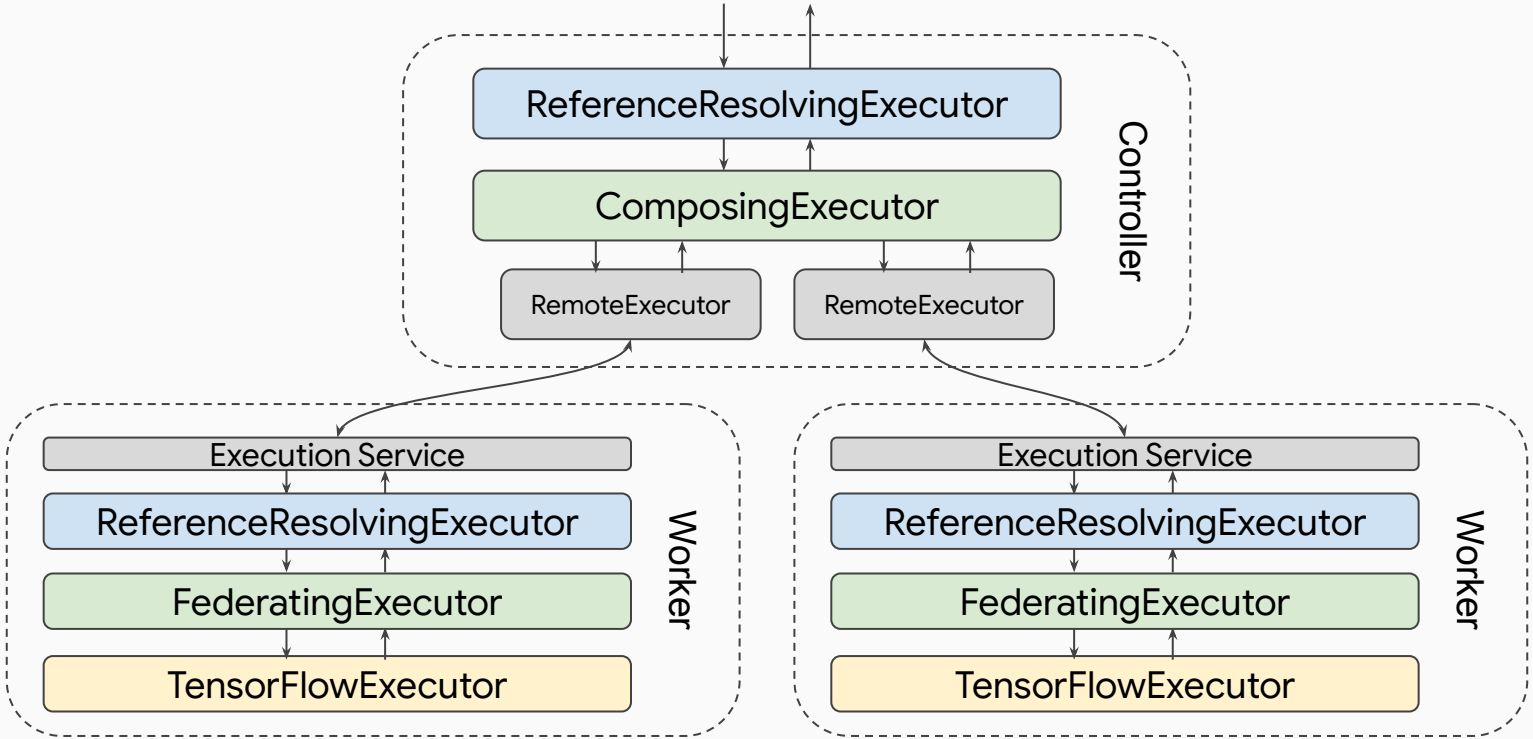
```
(mul_by_num_c_arg -> (let
  fc_mul_by_num_c_symbol_0=federated_broadcast(mul_by_num_c_arg),
  fc_mul_by_num_c_symbol_1=federated_sum(fc_mul_by_num_c_symbol_0)
 in fc_mul_by_num_c_symbol_1))
```

# Federated Intrinsics require a **FederatingExecutor**




This stack is enough to run full simulations within a single machine*

# Large cohorts execute too slow! Scale horizontally!



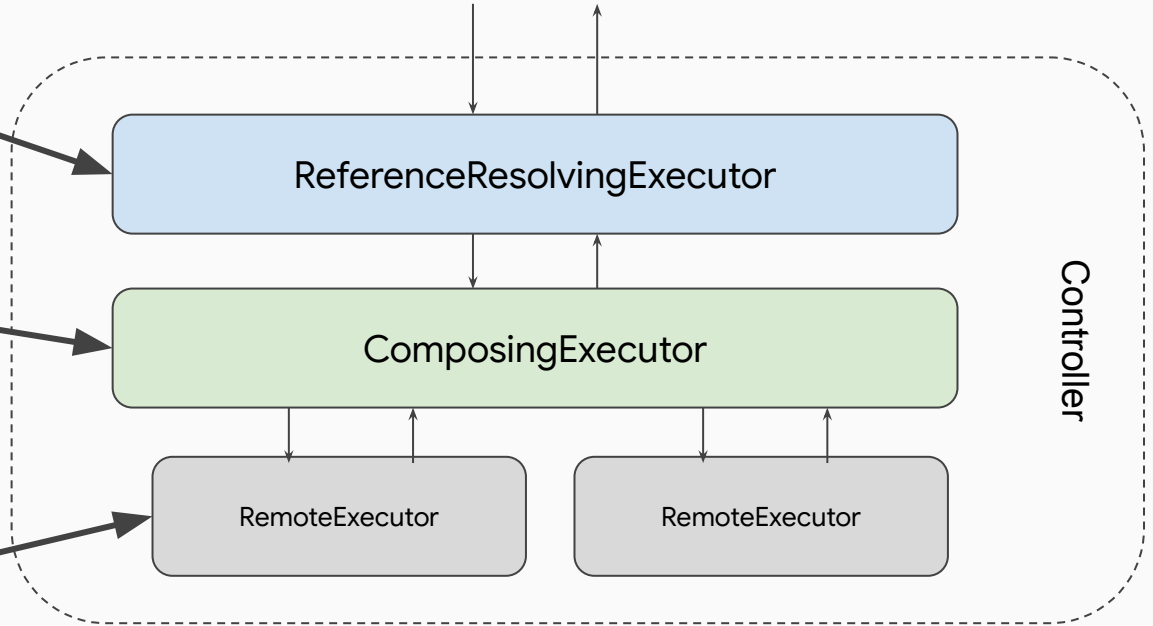

**Note**: the **FederatingExecutor** on the **Worker** is "packing" multiple clients in the cohort in a single machine. There maybe scenarios where only the **TensorFlowExecutor** is on the Worker and consequently have a single "client" per Worker.

# Deep dive: executor responsibilities

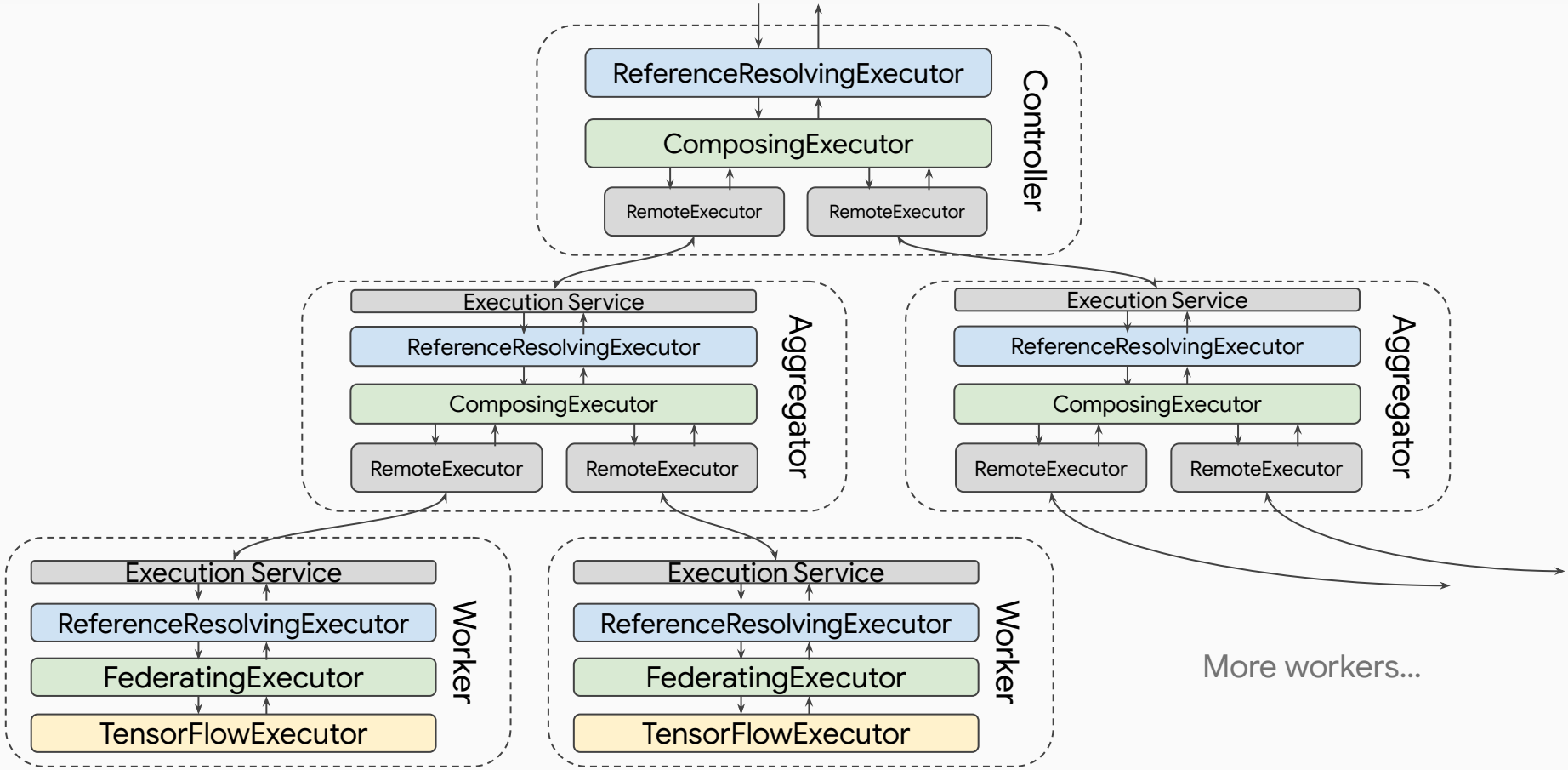
**Evaluates** logic in computation: **reduces** computations for its targets

**Partitions logic and arguments**: bridges from single executor to many executors

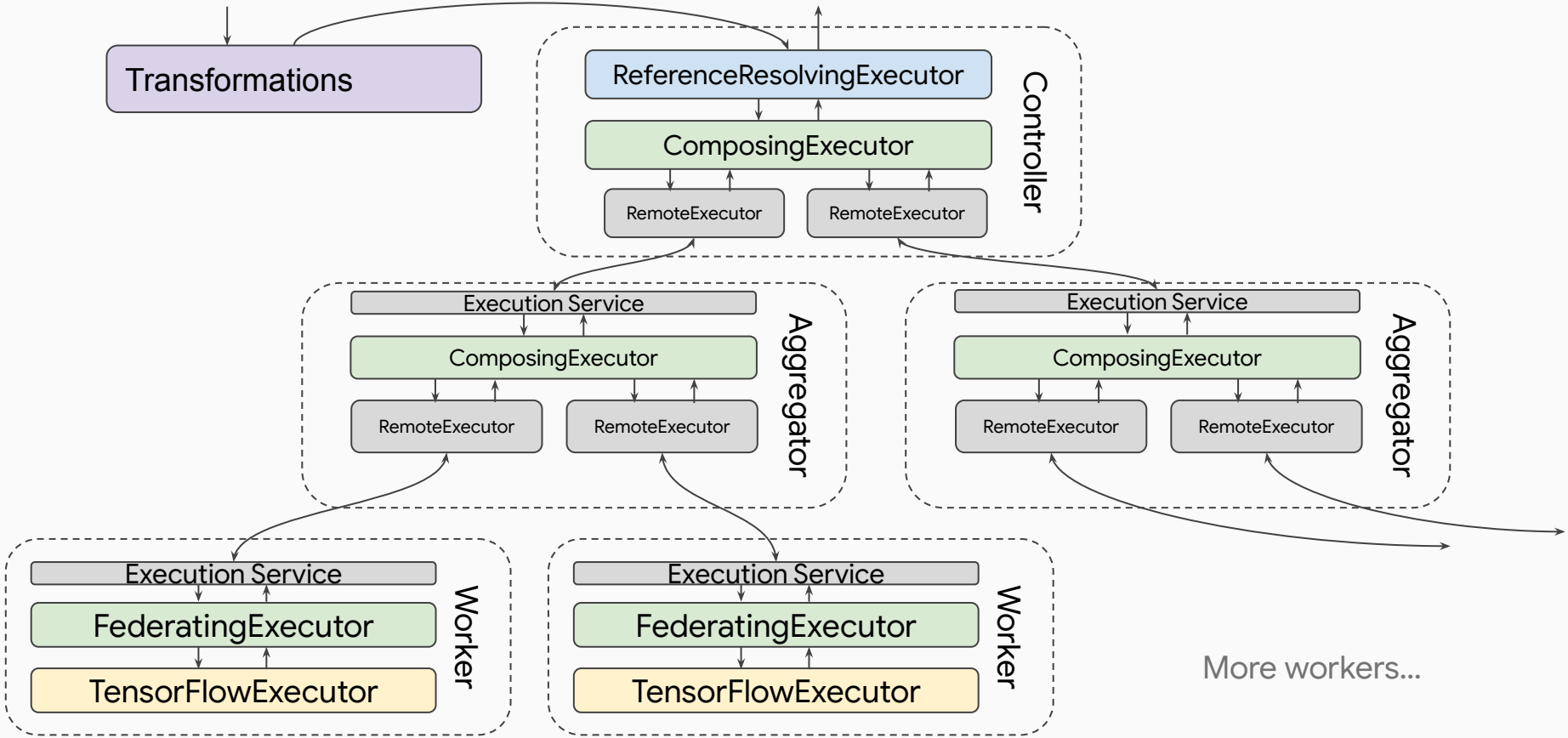
**Proxies for remote machine**: doesn't change the incoming computations at all!



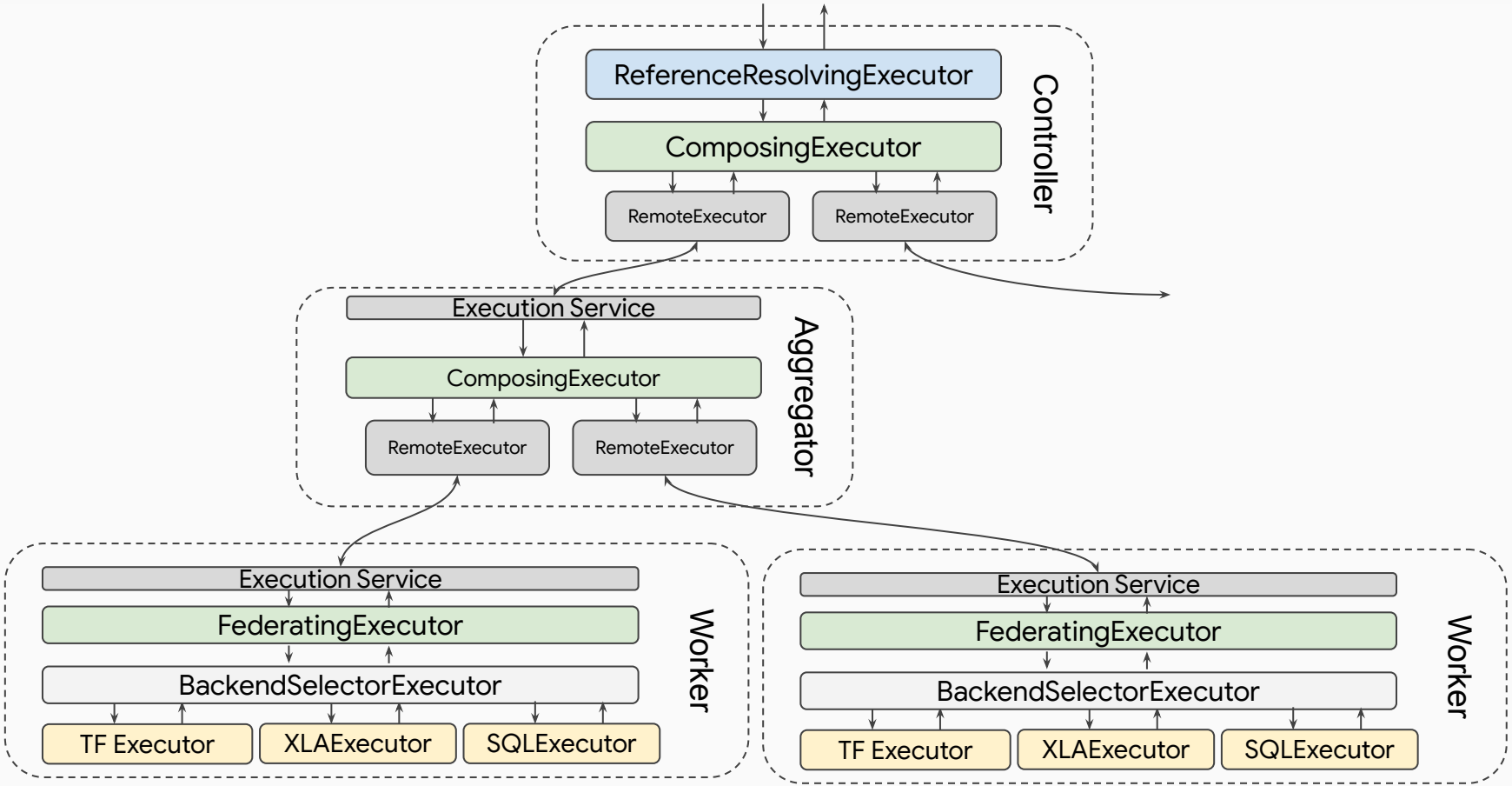
# Aggregation is too slow! Add hierarchical aggregation!



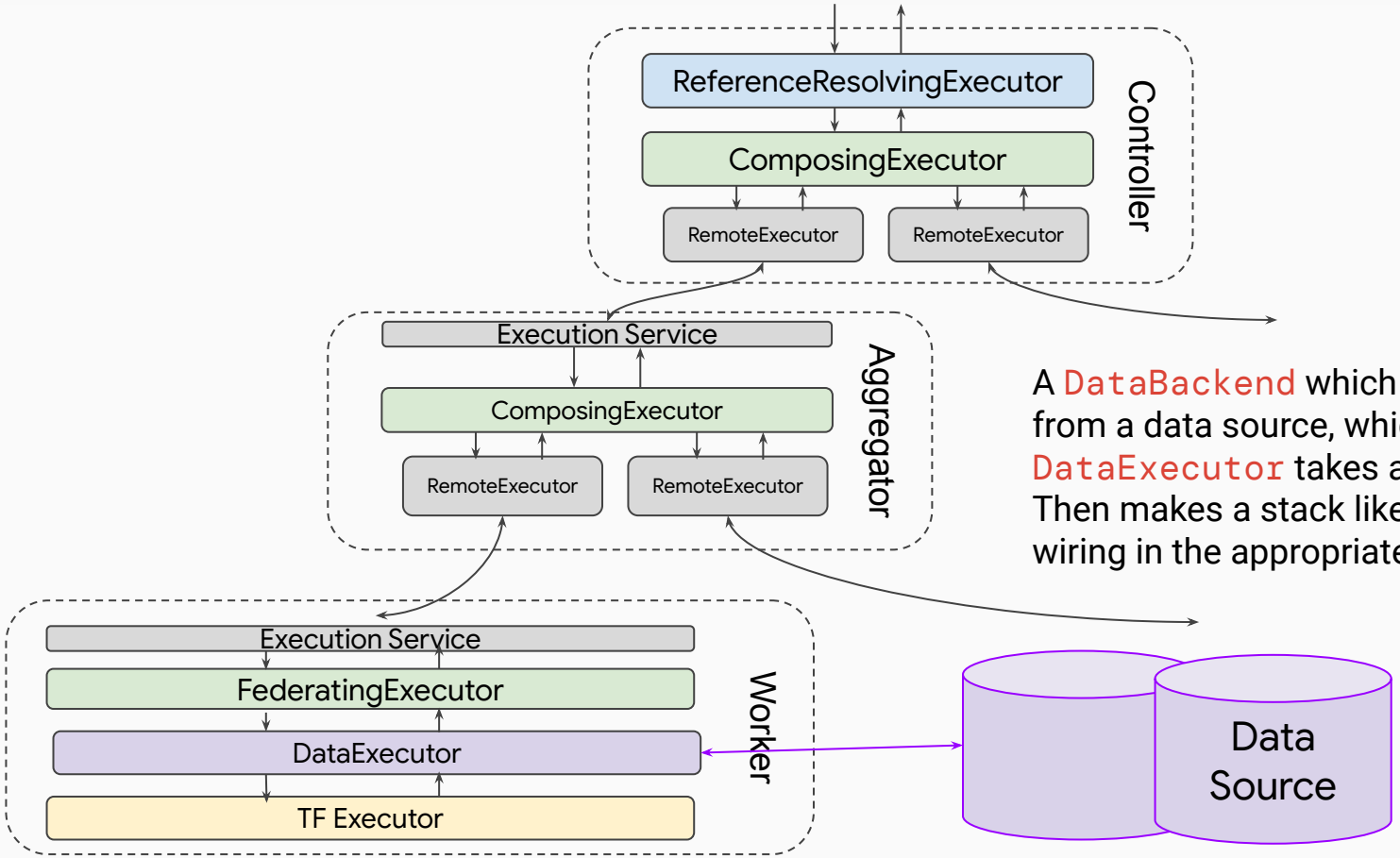
# Forward pointer to TFF-103:
# Transforming the `Computation` can change the executor stack requirements.



# What about extending to non-TF backends (e.g. JAX, or SQL)? Implement a new executor add to the bottom of the executor stack!

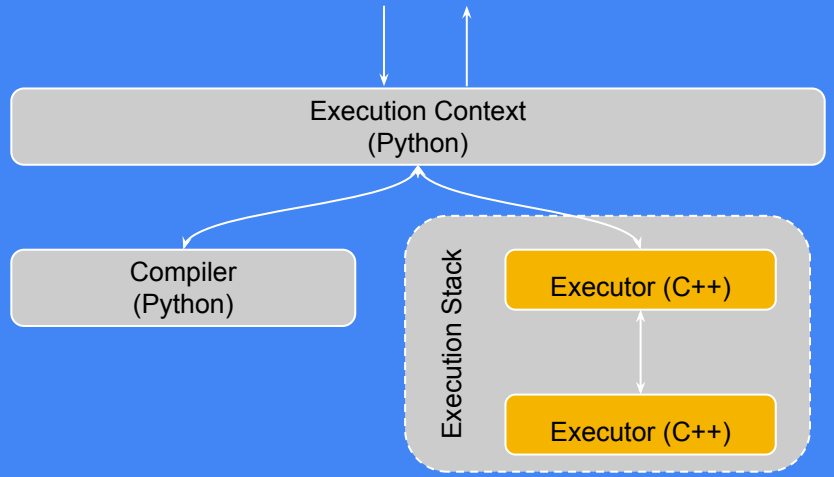


# How does add a new data backend?




A `DataBackend` which can read from from a data source, which the `DataExecutor` takes as a parameter. Then makes a stack like this picture, wiring in the appropriate `DataBackend`.

# Executor


Execution Context
(Python)
Compiler
(Python)
Execution Stack
Executor (C++)
Executor (C++)

# Executor

**Is an abstract interface.**

The C++ pure virtual interface is at:
https://github.com/tensorflow/federated/blob/main/tensorflow_federated/cc/core/impl/executors/executor.h

This interface has five cardinal methods:

1. `ValueId CreateValue(v0::Value)`
   Embeds a new Value in the executor and returns a handle to it.
2. `ValueId CreateCall(ValueId arg, ValueId function)`
   Calls a Value with an optional Value arguments and returns a handle to the resulting Value.
3. `ValueId CreateStruct(std::vector<ValueId> elements)`
   Creates a new struct Value from a list of values, returning the handle to the new struct Value.
4. `ValueId CreateSelection(ValueId struct, int32_t index)`
   Selects an element from a struct Value given an index, returns the handle to the struct element Value.
5. `Materialize(ValueId id, v0::Value* result)`
   Given a handle to a value, returns the concrete value.

# What happens during the invocation of x(5)?

```
@tff.tf_computation(tf.int32)
def add_one(x):
  return x + 1

x(5)
```

**Pseudo-code of invocation logic:**

```
function_handle = executor.CreateValue(
  add_one)

argument_handle = executor.CreateValue(5)

result_handle = executor.CreateCall(
  function_handle, argument_handle)

result = executor.Materialize(
  result_handle)

return result
```

# How do executors talk to each other?

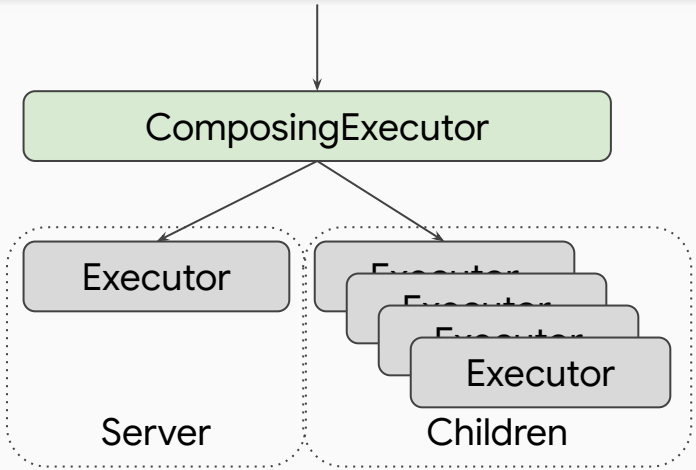
- Since `Executor` is an abstract interface, executors in general **don't know the capabilities of their children**.
- They only know what they themselves can do, and forward 'anything else' on down to its children.
  - Conceptually, this is similar to the "Chain of Responsibility" design pattern.
- Executors do this 'forwarding' **through the `Executor` interface.**

# Example: ComposingExecutor

Generally, the `ComposingExecutor`'s job is to 'partition work' to its children.

- These child executors are received by the `ComposingExecutor` at construction time. The `ComposingExecutor` **does not know their capabilities**. Often they are `RemoteExecutors`, proxies for executors hosted on other machines, though they do not need to be.

We will walk through 'logic splitting' in the case of two intrinsics: `tff.federated_map` and `tff.federated_aggregate`




```
shared_ptr<Executor> CreateComposingExecutor(
    shared_ptr<Executor> server,
    std::vector<shared_ptr<Executor>> children);
```

# ComposingExecutor::CallIntrinsicMap

`ComposingExecutor` iterates over its child executors, embedding in each of them:

- the map intrinsic
- the function being mapped
- a slice of the data argument

```
auto child_map = TFF_TRY(
  child->CreateValue(map_val));
auto child_fn = TFF_TRY(
  child->CreateValue(fn_val));
auto child_data = data.clients()[i]->ref();
```

This data has already been ‘partitioned’ into as many slices as there are children of the composing executor

# ComposingExecutor::CallIntrinsicMap

`ComposingExecutor` tuples together the mapping function and the data in its children.

Since `federated_map` is a function of two arguments: the mapping function and the data to be fed.

```
auto map_args = TFF_TRY(child->CreateStruct({child_fn, child_data}));
```

# ComposingExecutor::CallIntrinsicMap

- Finally, `ComposingExecutor` calls the `federated_map` in its child executors with the tupled together argument.
  - Then repackages the result, does some bookkeeping, etc., but we will skip today.
- `ComposingExecutor` now has a vector of **handles** to these mapped functions, called in its child executors. These it will repackage into a new piece of ‘distributed data’

```
auto result = TFF_TRY(child->CreateCall(child_map, map_args));
```

# ComposingExecutor::CallIntrinsicAggregate

This example is more exciting. To perform partial aggregations, `ComposingExecutor` **can't just directly forward all the calls!**

- Exercise for the reader: why not?

# Interlude: `ComposingExecutor` splitting up work while retaining semantics

Consider: `tff.federated_aggregate(value, zero, accumulate, merge, report)`

`merge` and `accumulate` we can run as many times as we want; no problem, pass it right down to our children

`report` we cannot! Consider, e.g., `federated_mean`; this step is *dividing the numerator by the denominator*! We can't just repeatedly do that!

# ComposingExecutor::CallIntrinsicAggregate

To handle this ComposingExecutor **constructs new** federated_aggregate **arguments on the fly**. These arguments will pass down to its children (next slide).

# ComposingExecutor::CallIntrinsicAggregate

```
v0:Value null_report_val;
*null_report_val.mutable_computation() = IdentityComp();
```

Constructed on the fly inside the ComposingExecutor!

The rest of the federated aggregate arguments come directly from the original arguments

```
for (const v0::Value& arg_value :
     {zero_val, *accumualte_val, *merge_val, null_report_val}) {
  OwnedValueId child_id = TFF_TRY(child->CreateValue(arg_value));
}
```

# ComposingExecutor::CallIntrinsicAggregate

Everything proceeds similarly as `CallIntrinsicMap`, tupling arguments, making calls to children executors, to compute the result of merging.

Finally, the **original** report function is **reinjected**, run by the composing executor itself, in the server-placed executor it owns.

```
auto report_id = TFF_TRY(report.unplaced()->Embed(*server_));

auto result =
  TFF_TRY(server_->CreateCall(report_id->ref(), current.value()));
```

# Wrapping up: What did we learn?

# TFF Execution: the big ideas

- Computation is made of **composable pieces**, each with its own ‘small chunk’ of meaning. This philosophy is reflected in the way it is interpreted: **composable pieces**, each responsible for some small chunk of behavior.
  - This design makes execution **easy to extend**. For example, adding additional backends (e.g. JAX/XLA) becomes a small task.

- **Flexible parameterization** of executor-to-user bridge (ExecutionContext) means behavior can be dynamically dispatched at multiple levels.
  - ExecutionContext has *global* computation information (can see the entire computation it’s invoking); it can then, for example, configure the runtime differently for computations with different properties.