Executing Cross-language Transforms in the Beam Go SDK

Kevin Sijo Puthusseri
(kevinsijo@google.com, @pskevin)

Uber JIRA

Note

This document is meant to be a sufficient not complete resource to understand the challenges and suggested solutions associated with supporting execution of cross-language transforms in the Go SDK. Thus, information assumed to be foreknowledge has been redundantly summarized to provide better context. Corresponding complete and detailed resources are linked wherever applicable.
It is a work in progress and represents only what I've understood. Thus, obviously, it may not accurately represent facts and requires everyone's input. Any and all opinions are absolutely required and invited.

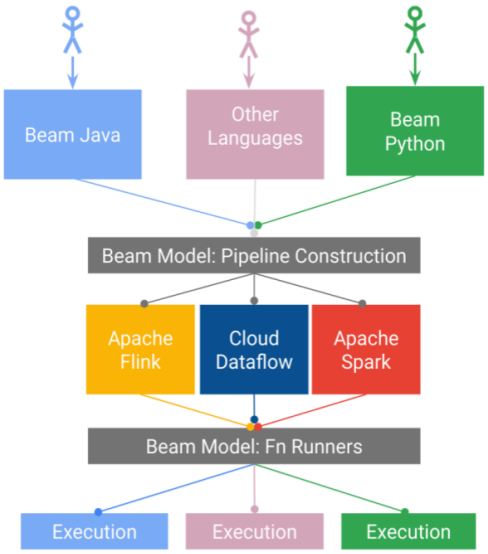
Goal

→ Add an **External Transforms client** that enables running Java/Python transforms from a pipeline authored using the Go SDK

→ Validate correctness by adding **Go SDK tests** to existing cross-language Flink/Spark test suites

Background

> *Note* | *All required foreknowledge has been described under their respective sections only to provide better context. Feel free to skip parts you understand.*



Beam's portability efforts allow, by design[1], for transforms authored in *any* supported SDK to be executed on *any* runner. The SDK-Runner boundaries through pipeline construction and execution, are linked by generic proto representations streamed over gRPC.

The decoupled phases can be summarized into:

1. **Pipeline Construction**
   The authoring SDK represents the pipeline as a language-agnostic proto which is submitted to a *portable* runner along with associated artifact(s).

2. **Pipeline Execution**
   The portable runner orchestrates execution by spawning workers (SDK harnesses) within environments as needed by the transforms.

Cross-language Pipelines

SDK Components

The Beam Fn API supports executing code written in arbitrary languages[2] thus enabling the cross-language pipeline execution phase to work out of the box. Constructing pipelines that use External transforms (i.e. transforms from foreign SDKs), on the other hand, is more involved than pipelines strictly using transforms from the authoring language because:

- → The authoring SDK doesn't understand how to decompose the external transform into a DAG of Beam's primitive transforms
- → The authoring SDK doesn't know upon which artifacts the external transform (decomposed or not) depends

Thus, to construct cross-language pipelines, SDKs need to be augmented with additional components.

---

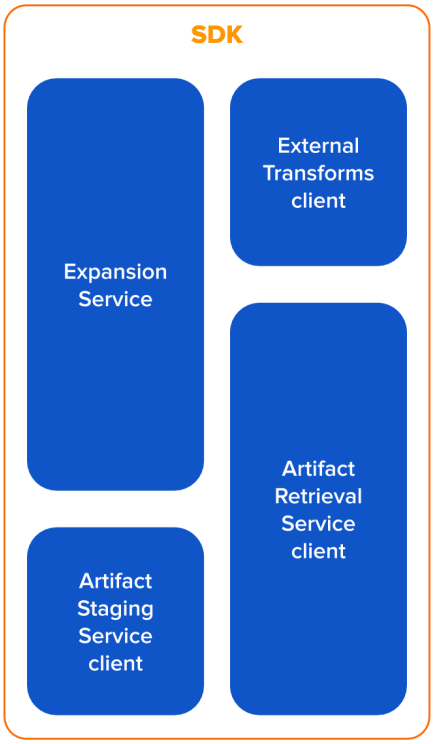
[1] *Portable Beam Overview* diagram sourced from Apache Beam Project Overview slide deck
[2] Excerpt from the Cross-language Beam Pipelines design document

Moreover, each SDK that supports cross-language transforms needs take into account that it may:

- ⟶ Use an external transform provided by a foreign SDK
- ⟶ Author an external transform used by other SDKs

> *Note* | *This design document proposes a solution to the first use case delineated above. The second use case will be discussed, in the future, in a different design document.*

The various cross-language components and their role within an SDK can be understood as follows:



⟶ **Expansion Service**
It is responsible for providing the correct representation and associated artifacts for transforms the SDK exposes to be consumed by foreign SDKs as external transforms.

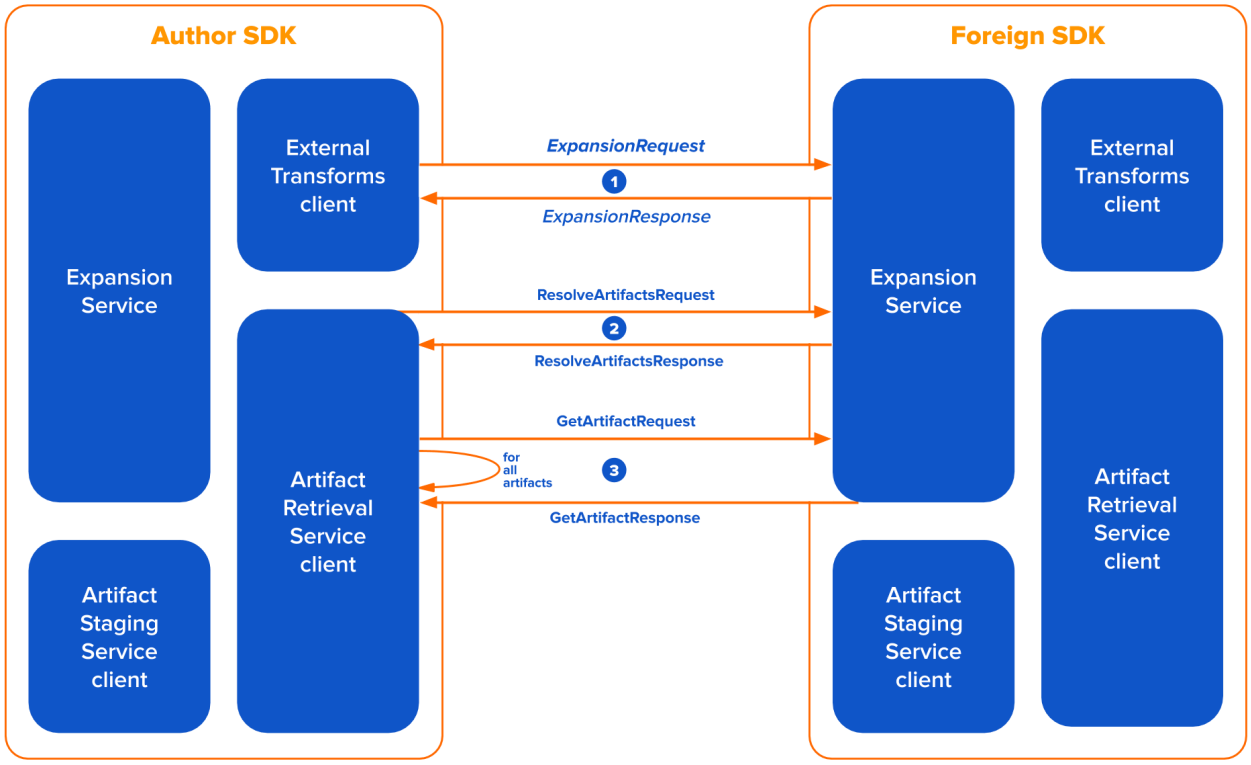
⟶ **External Transforms client**
It interfaces with the foreign SDK's Expansion Service for correct representation of an external transform. This representation is then added to the natively existing DAG of transforms eventually submitted as a pipeline proto.

⟶ **Artifact Retrieval Service client**
It resolves and stores locally artifacts required by the expanded external transform with the help of the foreign SDK's Expansion service. It is also used by the SDK harness to retrieve staged artifacts from the Job Service during pipeline execution.

⟶ **Artifact Staging Service client**

It relays locally stored artifacts to the Job Service during job submission. These staged artifacts are further used to instantiate correct environments for the workers before pipeline execution can begin.

Construction Flow



Pipeline construction of an External transform can be decomposed into the following steps:

1. **Transform Expansion**

```
message ExpansionRequest {
    Components components = 1;
    PTransform transform = 2;
    string namespace = 3;
}
```

```
message ExpansionResponse {
    Components components = 1;
    PTransform transform = 2;
    repeated string requirements = 3;
    string error = 10;
}
```

The pipeline authoring SDK's *External Transforms client* uses protos of the above form to query the foreign SDK's *Expansion service* for an expanded representation (comprising the DAG of Beam's primitive transforms) of the external transform. The expanded transform's *FunctionSpec*[3] will contain information regarding the environment and artifact(s) required to execute the transform.

[3] GitHub permalink to FunctionSpec and PTransform's FunctionSpec definitions in the Beam Runner API proto

2. **Artifact Resolution**

```
message ResolveArtifactsRequest {
  ArtifactInformation artifacts = 1;
  repeated string preferred_urns = 2;
}
```

```
message ResolveArtifactsResponse {
ArtifactInformation replacements = 1;
}
```

Information of the artifact(s) may be incomplete since they may only represent higher order dependencies. In order to get the list of transitive dependencies associated with this external transform, the authoring SDK's *Artifact Retrieval Service client* queries the foreign SDK's expansion service using the protos above[4].

3. **Sourcing Artifacts**

```
message GetArtifactRequest {
  ArtifactInformation artifact = 1;
}
```

```
message GetArtifactResponse {
  bytes data = 1;
}
```

Once the artifact information has been received, the authoring SDK's *Artifact Retrieval Service client* queries the foreign SDK's expansion service using the protos above, for the actual artifact. The authoring SDK then stores these artifacts locally and updates their corresponding *ArtifactInformation*[5] to reflect the local location.

Artifact Staging and Job Submission

TODO(kevinsijo): Visualizing *ReverseArtifactRetrievalService* in context to job submission.

The authoring SDK streams the local artifacts to the *Artifact Staging Service* using the staging token received from PrepareJob after connecting to the Job Service by implementing the *ReverseArtifactRetrievalService*.[6] The Artifact Retrieval Service uses a retrieval token from the provisioning service to get a registered list of artifacts. The workers then use the Artifact Retrieval Service client and stream the required artifacts from the Artifact Retrieval Service using the retrieval token received from the provisioning service.[7]

---

[4] Most recent implementation of Artifact Resolution from Heejong Lee's Cross-language Artifact Staging: Design on Implementation Details

[5] GitHub permalink to ArtifactInformation definitions in the Beam Runner API proto

[6] Github permalink to ReverseArtifactStagingService in the Beam Artifact API proto

[7] Most proposed changes to Pipeline submission from Heejong Lee's Cross-language Artifact Staging: Design on Implementation Details

Go SDK

Having examined how external transforms are part of pipelines authored in Java/Python SDKs, it is useful to understand how the Go SDK works to enable execution of external transforms in it as well. The Go SDK is notably different from the Java/Python SDKs since:

- → Absence of generics makes it rely heavily on *reflection* to ensure type consistency across the pipeline.
- → In order to monitor and handle composite transforms, it has a notion of "*scopes*" that associate PTransform(s) and PCollections together.
- → Helper constructs to memoize parts of the graph and handle *proto↔instance* conversions are segregated in different packages/functions.

Graph Construction

Adding an expanded external transform to the existing pipeline requires understanding how the pipeline is represented within the SDK. As of now, the entire pipeline exists as a hypergraph[8] where every PCollection is a node and each PTransform is a multi-edge.

The approximate form of the hypergraph and its associated types are given below:

```
type Graph struct {
    scopes []*Scope
    edges  []*MultiEdge
    nodes  []*Node
    root *Scope
}
```

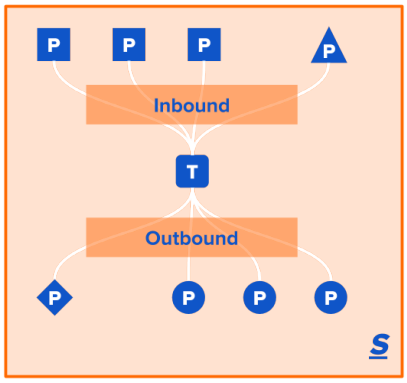
```
type Node struct {
    id int
    t typex.FullType
    Coder *coder.Coder
    w *window.WindowingStrategy
    bounded bool
}
```

```
type MultiEdge struct {
    id     int
    parent *Scope
    Op     Opcode
    /*Operation related fields*/
    Input  []*Inbound
    Output []*Outbound
}
```

```
type Inbound struct {
    Kind InputKind
    From *Node
    Type typex.FullType
}
```

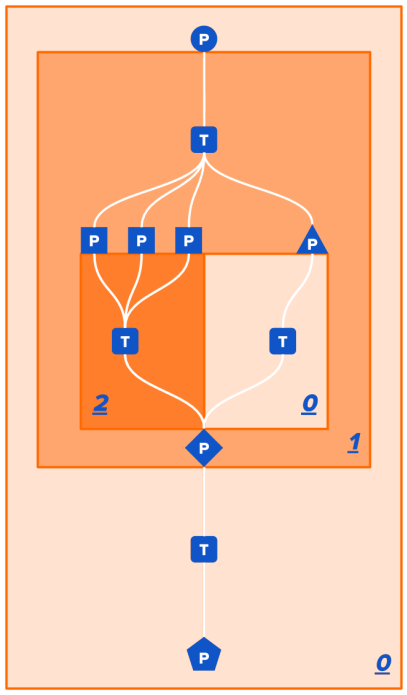
```
type Outbound struct {
    To *Node
    Type typex.FullType
}
```

[8] A Hypergraph is a generalization of a graph in which an edge can join any number of vertices.



Each `MultiEdge` is an edge in the hypergraph's edge and represents a PTransform as depicted in the adjoining figure. Each `Node` (**P**) represents a single PCollection. `Inbound` and `Outbound` represent the anticipated incoming and outgoing types of PCollections respectively. The `MultiEdge` is bounded by a `Scope` (**S**). Organizing scopes as parents and children help in composing and monitoring composite transforms.

> *Note* *Multiple types of `Inbound` links represent the possibility of multiple inputs (main and side).*
> *Multiple types of `Outbound` links represent the possibility of multiple emitters within the same PTransform.*



This figure above exemplifies how the Go SDK represents a pipeline with PTransforms that operate on multiple inputs and emit multiple outputs as a hypergraph organized contextually by a hierarchy of scopes.

Handling Artifacts

Components that manage artifacts exists independently, siloed in different packages:

- ⟶ `translate.go`[9] in `beam/core/runtime/exec/` translates *proto⟶instance*
- ⟶ `translate.go`[10] in `beam/core/runtime/graphx` translates *instance⟶proto*
- ⟶ `materialize.go`[11] in `beam/artifact` implements the *Artifact Retrieval Service client*
- ⟶ `stage.go`[12] in `beam/runners/universal/runnerlib` implements the *Artifact Staging Service client*

[9] proto⟶instance
[10] instance⟶proto
[11] Artifact Retrieval Service client
[12] Artifact Staging Service client

Proposal

External Transforms API

A barebones cross-language transforms client generally has the following API:

```
function External (string urn, byte[] payload, string expansionAddr) {...}
```

In order to support cross-language transforms in the Go SDK, a new API is proposed and is contrasted with the existing API as follows:

Current API	Proposed API

`func External(`
`  s  Scope,`
`  spec  string,`
`  payload []byte,`
`  in  []PCollection,`
`  out  []FullType,`
`  bounded bool,`
`) []PCollection`
	`func External(`
	`  s  Scope,`
	`  urn  string,`
	`  payload  []byte,`
	`  in  []PCollection,`
	`  out  []FullType,`
	`  bounded  bool,`
	`  expansionAddr string,`
	`  opts  ...Option,`
	`) []PCollection`

- → As explained previously, `s Scope` is necessary and thus, is retained.
- → `spec string` is renamed to `urn string` to better reflect its relevance.
- → `payload []byte` is retained since it's required by the *Expansion Service*.
- → `in []PCollection` is retained since it's required to populate `ExpansionRequest` proto.
- → Type inference of the output PCollections from the expanded external transform isn't possible since reverse schema decoding is still being implemented by Robert Burke. Due to this, for sake of accurate pipeline construction, the current API requires the anticipated output type(s) to be explicitly passed. For the same reason and backwards compatibility, the proposed API will retain the `out []FullType` field.
- → `bounded bool` is retained to further specify whether the output is bounded or not.
- → `expansionAddr string` is added to query the *Expansion Service*.
- → In contrast to the existing API, the variadic `opts ...Option`[13] field is added since multiple inputs (main and/or side) are expected from a cross-language transforms client.

[13] Option interface from `beam/option.go`

Handling Artifacts

As of now, the Go SDK constructs environments corresponding to each PTransform when the entire graph is converted into proto just before job submission using `translate.go` in `beam/core/runtime/graphx`. A Go environment is added to each PTransform by default since, obviously, they are all natively in Go.

To support cross-language transforms, this is not feasible anymore. Artifacts belonging to each PTransform part of the expanded external transform need resolution before pipeline submission. Moreover, in contrast to the Java/Python SDKs, this proposal explicitly chooses to defer artifact resolution to *instance→proto* translation. This is to avoid wasteful network consumption caused by resolution and fetching of artifacts everytime pipeline construction fails due to erroneous user code. Regardless of when artifact resolution occurs, the environments received as *ExpansionResponse* need to be saved and made referenceable to the correct PTransform for further processing.

Thus, changes to `Graph` are proposed and reasoned as follows:

Current API	Proposed API

`type Graph struct {`
`  scopes []*Scope`
`  edges []*MultiEdge`
`  nodes []*Node`
`  root *Scope`
`}`
	`type Graph struct {`
	`  scopes []*Scope`
	`  edges []*MultiEdge`
	`  nodes []*Node`
	`  root *Scope`
	`  envs map[int]*pipepb.Environment`
	`}`

- → Each `MultiEdge` has a unique `id int` associated with it. Using this, for each PTransform of the expanded external transform, it's associated environment can be saved as `envs map[int]*pipepb.Environment` to be referenced and resolved later.
- → For every transform whose `id` is a part of the `envs` field, during *instance→proto* translation, artifacts will be resolved and fetched.
- → Each `pipepb.Environment` with a local artifact will be updated to reflect correct paths using which they will be staged during job submission.

Construction Flow

1. **User call**

```
out := beam.External(s, urn, payload, in  out, bounded, expansionAddr, opts)
```

2. **Proposed v/s Legacy**
   If `expansionAdd` or `opts` fields are present, then the proposed API will be called otherwise the legacy API will be called with its expected arguments

3. **Expansion**

   The external transform will be expanded by querying the *Expansion Service* and added to the existing graph using:
   - → Helper functions from `translate.go` in `beam/core/runtime/exec/` and `beam/core/runtime/graphx` to handle *proto↔instance* translation required to:
     - ⇒ Build the `ExpansionRequest`
     - ⇒ Interpret the `ExpansionResponse` and store `Environment` for each PTransform of the expanded external transform
   - → Helper functions from `edge.go`[14] in `beam/core/graph` to add the expanded transform and it's subgraph to the existing hypergraph.

4. **Handling Artifacts**

   During *instance→proto* translation, artifacts for each environment in `Graph.envs` are resolved and fetched locally using:
   - → Helper functions from `materialize.go` in `beam/artifact` for artifact resolution and retrieval
   - → Environments for each retrieved artifact are updated to reflect local paths

Organization and Dependency

The external transforms API and expansion related code will reside in `external.go`[15] under `beam/`. In the future, when Go SDK's *Expansion Service* is implemented in `expand.go`, it will also be under `beam/`.
Ideally, *proto↔instance* translation should be abstracted out into a common library package.

Convenience Wrappers

It would be really convenient for users if the types for output PCollections are wrapped around and declared in advance.
Obviously, IO connectors need to provide convenience wrappers that do more than just this.

---

[14] Hypergraph helper functions
[15] External Transforms API

Validating Cross-language Go SDK

TODO(kevinsijo): Describe design for this, when the previous parts are implemented.

```
func TryCrossLanguage(s Scope, ext *graph.ExternalTransform)
(map[string]*graph.Node, error
```

Appendix

Glossary

```
func TryCrossLanguage(s Scope, ext *graph.ExternalTransform) (map[string]*graph.Node, error) {
	// Add ExternalTransform to the Graph

	// Validating scope
	if !s.IsValid() {
		return nil, errors.New("invalid scope")
	}

	// Using existing MultiEdge format to represent ExternalTransform (already backwards compatible)
	edge := graph.NewCrossLanguage(s.real, s.scope, ext)

	// Build the ExpansionRequest

	// Obtaining the components and transform proto representing this transform
	p, err := graphx.Marshal([]*graph.MultiEdge{edge}, &graphx.Options{})
	if err != nil {
		return nil, errors.Wrapf(err, "unable to generate proto representation of %v", ext)
	}

	xlangx.AddFakeImpulses(p)

	// Assembling ExpansionRequest proto
	transforms := p.GetComponents().GetTransforms()
```

```
	rootTransformID := p.GetRootTransformIds()[0] // External transform is the only root transform
	rootTransform := transforms[rootTransformID]

	delete(transforms, rootTransformID)

	req := &jobpb.ExpansionRequest{
		Components: p.GetComponents(),
		Transform:  rootTransform,
		Namespace:  s.String(), //TODO(pskevin): Need to be unique per transform (along with the UniqueName which is just string(Opcode) for now)
	}

	res, err := xlangx.Expand(context.Background(), req, ext.ExpansionAddr)
	if err != nil {
		return nil, errors.WithContextf(err, "failed to expand external transform with error [%v] for ExpansionRequest: %v", res.GetError(), req)
	}

	xlangx.RemoveFakeImpulses(res.GetComponents(), res.GetTransform())

	exp := &graph.ExpandedTransform{
		Components_:     res.GetComponents(),
		Transform_:      res.GetTransform(),
		Requirements_:   res.GetRequirements(),
		BoundedOutputs_: make(map[string]bool),
	}
	ext.Expanded_ = exp

	xlangx.VerifyNamedOutputs(ext)

	xlangx.ResolveOutputIsBounded(ext)

	// Using information about the output types and bounded nature inferred or explicitly passed by the user
	graph.AddOutboundLinks(s.real, edge)

	return ext.Outputs, nil
}

```