

# ArcGIS Velocity

## gRPC Client Developer Guide

Essential tasks and best practices for getting started with developing client applications for the gRPC feed in ArcGIS Velocity

Publication date: December 4, 2021



## Contents

Getting started with the gRPC Feed in ArcGIS Velocity .....	2
gRPC Implementation .....	2
gRPC service methods.....	2
gRPC performance .....	2
Key gRPC feed concepts.....	2
Usage notes.....	4
Server Implementation .....	5
Create the feed in ArcGIS Velocity.....	5
Create a feed using the ArcGIS Velocity Swagger API.....	6
Client Implementation .....	10
Create the boilerplate code .....	10
Methods for sending data.....	11
Response send(Request) .....	11
void send(Request, StreamObserver) .....	11
StreamObserver Stream(StreamObserver) .....	11
Flow Control.....	11
Feature Wrapper.....	12
Error handling .....	12
gRPC Response Codes .....	12
Appendix .....	15
Host name .....	<b>Error! Bookmark not defined.</b>
ArcGIS Velocity Swagger API .....	15

## Getting started with the gRPC Feed in ArcGIS Velocity

The [gRPC feed type](#) in ArcGIS Velocity creates a built-in gRPC server endpoint that can be used to programmatically send data from a remote gRPC client application to Velocity with gRPC requests. Client applications can be developed in a variety of languages, as supported by the [gRPC framework](#). Utilizing the tools provided by the gRPC framework, the underlying communication stub objects required to communicate with Velocity are generated automatically. Developers can create the custom client applications around the provided stubs.

### gRPC Implementation

Those who are not new to gRPC or read the gRPC documentation prior to reading this document should be aware of the following concepts that are implemented by the Velocity gRPC Feed.

#### gRPC service methods

Within gRPC, the Service definition allows the definition of four service methods: Unary RPCs, Server Streaming RPCs, Client Streaming RPCs, and Bidirectional RPCs.

<https://grpc.io/docs/what-is-grpc/core-concepts/#rpc-life-cycle>

Velocity's gRPC Feed implements both of the following:

- **Unary RPC:** The client sends a single request to the server and gets a single response back, just like a normal function call.
- **Client streaming RPC:** The client writes/sends a sequence of messages to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.

The Velocity gRPC Feed has not implemented the Server Streaming RPC or Bidirectional RPC service methods.

#### gRPC performance

Before developing your own gRPC Feed client, please be sure to read through the [performance best practices](#) on the gRPC.io website. Specifically, please note the language specific suggestions for obtaining best performance.

<https://grpc.io/docs/guides/performance/>

### Key gRPC feed concepts

The ArcGIS Velocity gRPC feed type is built around several key concepts that are important to understand prior to developing a custom gRPC client for Velocity. Fundamentally, the gRPC feed type implements the standard requests/response remote procedure call defined by the gRPC framework. The details on how the standard communications have been implemented are below:

- **Feature:**
  - The Feature is synonymous with an event. It represents a single message to be sent to Velocity.
  - A Feature is defined as an array of Any values. Each data type (string, integer, long, float, double, Boolean, and more) has a translation method to convert it to the Any type.

- The order of the Feature's value array represents the pre-determined schema that is registered with the gRPC feed. In other words, when creating a Feature object, the order the values are added must match the schema the Velocity gRPC feed is expecting.
- The client message is typically wrapped with an adapter that will translate the client message into a Feature object. This ensures the client messages are properly translated into the event schema expected by the Velocity gRPC feed.
- Request:
  - The Request contains an array of one or more Features.
  - The Request is typically used as is.
- Response:
  - The Response contains a response code (like HTTP response codes) and a response message. The code and the message can be used to determine if the request (or stream of requests) was received successfully or if an error occurred.
  - A Response is returned as a result of each `send()` procedure call (one request, one response).
  - A Response is returned when a `stream()` procedure call is closed (many requests, one response).
  - The Response is typically used as is.
- Channel:
  - The Channel represents a communication channel opened between a client application and the Velocity gRPC feed.
  - Channels must always utilize transport security (SSL).
  - A Channel is created by providing a host name and a port.
    - Host name is a public fully qualified domain name of the hosting server. It is provided on the Velocity gRPC feed details page.
    - Port will always be **443**.
  - Channels are augmented with Metadata (see below) to assist with communication header values.
- Metadata:
  - The Metadata represents the underlying header values used when creating the Channel.
  - Feed path (required)
    - The header key **grpc-path** is required and must be set.
    - The header value for the **grpc-path** is the Velocity gRPC feed path provided on the Velocity gRPC feed details page.
  - Token authorization (optional)
    - The header key **authorization** can be provided if the Velocity gRPC feed has been created using ArcGIS authentication.
    - The header value, if set, will be `Bearer <token>`.
    - The token value is obtained from the sharing interface of the ArcGIS Online organization that is hosting ArcGIS Velocity. Typically, at:  
`https://<hostserver>.arcgis.com/sharing/rest/generateToken`
- Feed Stubs:

- The Feed Stubs decorate the underlying communications and present the standard methods for sending requests and receiving responses.
- Feed Stubs are created by passing in the Channel and Metadata information to the factory methods provided by the underlying framework.
- Methods:
  - Send:
    - A blocking RPC call to the server. The method sends a single Request and waits for a Response.
  - Stream:
    - A non-blocking call to open a channel to the server.
    - A StreamObserver is used to pass Requests to the open stream.
    - The server will not send a Response for each individual Request.
    - When a stream is closed, the server will send a single Response for the entire stream session.
- StreamObserver:
  - The StreamObserver is responsible for managing an open stream. It provides methods for sending requests, managing responses, and closing the stream. There are multiple implementations of the StreamObserver, depending on your needs.
    - Pass in a Response StreamObserver to the `send()` method to asynchronously process the response.
    - Pass in a Response StreamObserver to the `stream()` method to asynchronously monitor a streaming feed.
    - StreamObserver is returned from the `stream()` method to allow sending of Requests via the `onNext()` method.
  - Methods:
    - OnNext:
      - Passes the Request to the underlying channel to be sent to the server.
      - Receives single asynchronous Response for `stream()` when stream is closed.
      - Receives asynchronous Response for `send()`.
    - OnCompleted:
      - Closes the stream.
    - OnError:
      - Indicates an error was encountered on the underlying channel.

### Usage notes

- The Channel must always use transport security (SSL).
- The Metadata must always include the header key **grpc-path** with the value set to the path provided on the Velocity gRPC feed details page.
- When using ArcGIS security, the Metadata should include the header key **authorization** with the value set to the token obtained from the ArcGIS Online organization's `/sharing/rest/generateToken` endpoint.
- The Response code can either be the standard HTTP response codes (for example, 200 for success) or those reported by the gRPC API (see table below).

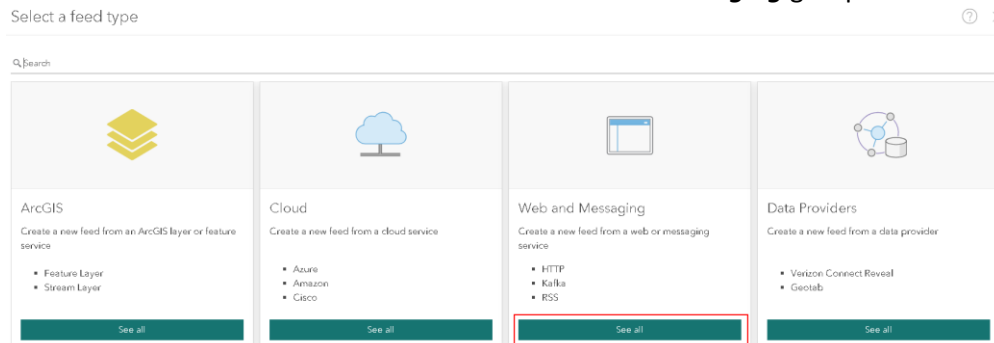
**EXAMPLE:** If the gRPC feed is stopped or the **grpc-path** is not set correctly, the following error will be observed:

[io.grpc.StatusRuntimeException](#): UNIMPLEMENTED: HTTP status code 404

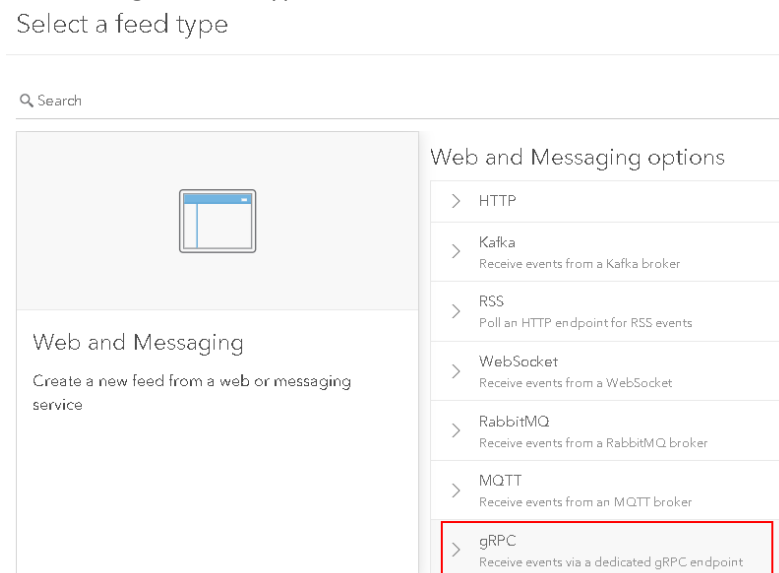
## Server Implementation

### Create the feed in ArcGIS Velocity


1. Click **Create feed** and click **See all** under the **Web and Messaging** group.



2. Select the **gRPC** feed type from the list.



3. Chose the authentication type.
  - a. **None** – No authentication is required to send data to the feed. This is not recommended.
  - b. **ArcGIS** – Provides token-based authentication. Currently, tokens are obtained by the client application using a user's credentials. The user must have access to ArcGIS Velocity and be a [Creator](#) user type.
4. Add the data schema.
  - a. Select the format of your data sample:
    - i. **Delimited** – comma, tab, or otherwise separated fields.
    - ii. **JSON**
    - iii. **XML**
  - b. Paste the data sample into the text field.

- i. Use **Format** and **Word Wrap**  to modify how the data sample is displayed.

Data format

Delimited

Derive schema

```

1 ReportID,DeviceID,Event,Alias,Email,FixTime,EventTime,Latitude,Longitude,AltitudeM,AccuracyM,SpeedKph,HeadingDeg,SignalStrengthDbms,
2 IsEmergency,IsPoweredOn
3 1,Route01,1,Ski Patrol 01,SkiPatrol01@telluride.com,2021-06-15,2021-06-15,37.93601798200,-107.81397831300,0.00000000000,0.30000000000,4.
4 00000000000,0.00000000000,-102.70000000000,False,True
5 2,Route0101,1,Ski Patrol 0101,SkiPatrol0101@telluride.com,2021-06-15,2021-06-15,37.93501033050,-107.84134993800,0.00000000000,2.
6 10000000000,11.90000000000,299.35000000000,-89.70000000000,False,True

```

- c. Click **Derive schema** to generate the data schema.

Data format

Delimited

Derive schema

Field Type	Field Name	Sampled Values
<input checked="" type="checkbox"/> Int32	ReportID	1 2
<input checked="" type="checkbox"/> String	DeviceID	Route01 Route0101
<input checked="" type="checkbox"/> Int32	Event	1 1
<input checked="" type="checkbox"/> String	Alias	Ski Patrol 01 Ski Patrol 0101
<input checked="" type="checkbox"/> String	Email	SkiPatrol01@telluride.com SkiPatrol0101@telluride.com
<input checked="" type="checkbox"/> String	FixTime	2021-06-15 2021-06-15
<input checked="" type="checkbox"/> String	EventTime	2021-06-15 2021-06-15
<input checked="" type="checkbox"/> Float64	Latitude	37.936017982 37.9350103305
<input checked="" type="checkbox"/> Float64	Longitude	-107.813978313 -107.841349938
<input checked="" type="checkbox"/> Float64	AltitudeM	0 0
<input checked="" type="checkbox"/> Float64	AccuracyM	0.3 2.1
<input checked="" type="checkbox"/> Float64	SpeedKph	4 11.9

Next

- d. Modify the field names and/or turn fields on/off if necessary, then click **Next**.
- e. Select your **Location**, **Time**, and **Track ID** fields, and click **Next**.
- f. Finally, add a **Name** and **Description** for the feed and click **Complete** to create the feed.
5. On the feed details page, review the new feed's properties. Of these, a few are important for gRPC client application development.
- gRPC endpoint header path** – Use the value of this parameter in the gRPC client to identify the feed the client is trying to communicate with.
  - gRPC endpoint URL** – Use this as the server host name the gRPC client will connect to.
  - Schema details** – The list of fields, their types, and the order they are expected. The gRPC client application is required to match this structure exactly when sending features to the gRPC feed.

## Create a feed using the ArcGIS Velocity Swagger API

Below is the JSON format for creating a new ArcGIS Velocity gRPC feed type.

- On the ArcGIS Velocity Swagger API page, locate the **Feed** section and expand the **POST/iot/feed/Creates a new feed instance** item.
- Copy and paste the updated JSON into the space provided and click **Execute**.

3. In the Velocity application, the new feed's details page should be available. See step 5 in the previous section for a description of the important properties.

```
{
  "id": "",
  "label": "Factual gRPC",
  "description": "A feed to receive Factual gRPC events",
  "feed": {
    "name": "grpc",
    "formatName": "grpc-feature",
    "schemaTransformation": {
      "inputSchema": {
        "attributes": [
          {
            "name": "id",
            "dataType": "Int64",
            "nullable": true,
            "tags": [
              "TRACK_ID"
            ]
          },
          {
            "name": "severity",
            "dataType": "Int64",
            "nullable": true,
            "tags": []
          },
          {
            "name": "factual_url",
            "dataType": "String",
            "nullable": true,
            "tags": []
          },
          {
            "name": "source_url",
            "dataType": "String",
            "nullable": true,
            "tags": []
          },
          {
            "name": "content",
            "dataType": "String",
            "nullable": true,
            "tags": []
          },
          {
            "name": "created_date",
            "dataType": "String",
            "nullable": true,
            "tags": []
          },
          {
            "name": "updated_date",
            "dataType": "Date",
            "nullable": true,
            "tags": [
              "START_TIME"
            ]
          },
          {
            "name": "arc_id",
            "dataType": "Int64",
            "nullable": true,
            "tags": []
          },
          {
            "name": "arc_name",
            "dataType": "String",
            "nullable": true,
            "tags": []
          }
        ]
      }
    }
  }
}
```



```

    },
    {
      "name": "latitude",
      "dataType": "Float64",
      "nullable": true,
      "tags": []
    },
    {
      "name": "longitude",
      "dataType": "Float64",
      "nullable": true,
      "tags": []
    },
    {
      "name": "location_name",
      "dataType": "String",
      "nullable": true,
      "tags": []
    },
    {
      "name": "location_type",
      "dataType": "String",
      "nullable": true,
      "tags": []
    },
    {
      "name": "regions",
      "dataType": "String",
      "nullable": true,
      "tags": []
    },
    {
      "name": "tags",
      "dataType": "String",
      "nullable": true,
      "tags": []
    },
    {
      "name": "verticals",
      "dataType": "String",
      "nullable": true,
      "tags": []
    }
  ],
  "geometry": {
    "geometryType": "esriGeometryPoint",
    "spatialReference": {
      "wkid": 4326
    },
    "hasZ": false,
    "hasM": false
  },
  "time": {
    "timeType": "instant"
  }
},
"fieldMappings": [
  {
    "fromField": "id",
    "toField": "id",
    "tags": [
      "TRACK_ID"
    ]
  },
  {
    "fromField": "severity",
    "toField": "severity",
    "tags": []
  },
  {
    "fromField": "factual_url",

```

```

        "toField": "factual_url",
        "tags": []
    },
    {
        "fromField": "source_url",
        "toField": "source_url",
        "tags": []
    },
    {
        "fromField": "content",
        "toField": "content",
        "tags": []
    },
    {
        "fromField": "created_date",
        "toField": "created_date",
        "tags": []
    },
    {
        "fromField": "updated_date",
        "toField": "updated_date",
        "tags": [
            "START_TIME"
        ]
    },
    {
        "fromField": "arc_id",
        "toField": "arc_id",
        "tags": []
    },
    {
        "fromField": "arc_name",
        "toField": "arc_name",
        "tags": []
    },
    {
        "fromField": "latitude",
        "toField": "latitude",
        "tags": []
    },
    {
        "fromField": "longitude",
        "toField": "longitude",
        "tags": []
    },
    {
        "fromField": "location_name",
        "toField": "location_name",
        "tags": []
    },
    {
        "fromField": "location_type",
        "toField": "location_type",
        "tags": []
    },
    {
        "fromField": "regions",
        "toField": "regions",
        "tags": []
    },
    {
        "fromField": "tags",
        "toField": "tags",
        "tags": []
    },
    {
        "fromField": "verticals",
        "toField": "verticals",
        "tags": []
    }
]

```

```

    },
    "properties": {
      "grpc-feature.yField": "latitude",
      "grpc-feature.xField": "longitude",
      "grpc-feature.dateFormat": "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'",
      "grpc-feature.buildGeometryFromFields": true,
      "grpc.authenticationType": "none"
    }
  },
  "properties": {
    "executable": true
  },
  "status": {
    "startTime": 1625853385613
  }
}

```

## Client Implementation

### Create the boilerplate code

The client will require boilerplate code to establish the gRPC objects. Each language is a bit different in how this is accomplished, but the pattern is generally the same. Using the ArcGIS Velocity [grpc\\_feed.proto](#) file, a tool provided by the gRPC API is run to generate the code. For more information on the supported languages, see <https://grpc.io/docs/languages/>.

These code files have already been created for some commonly used languages and can be [downloaded](#) for use in creating gRPC clients. Additionally, there are [reference implementations](#) that illustrate how to create a gRPC client using these libraries.

#### grpc\_feed.proto file:

```

syntax = "proto3";

import "google/protobuf/any.proto";

option java_multiple_files = true;
option java_package = "com.esri.realtime.core.grpc";
option java_outer_classname = "GrpcProto";

service GrpcFeed {
  // client streaming rpc for high velocity
  rpc stream(stream Request) returns (Response);
  // simple rpc for lower velocity
  rpc send(Request) returns (Response);
}

// main api methods
message Request {
  repeated Feature features = 1;
}

message Feature {
  repeated google.protobuf.Any attributes = 1;
}

message Response {
  string message = 1;
  int32 code = 2;
}

```

## Methods for sending data

The exact methods used to send data will be a bit different for each language, but this section provides a high-level overview of methods used to send data (this documentation is based on the Java stubs).

### `Response send(Request)`

The Synchronous / Blocking request `send(Request)` will send a single Request object, containing one or more Features, and block the calling thread until a response is returned. Use this method when a response from the server is required for every request and processing the response is required before the next Request is sent. This can be useful in situations where a queue of Features is maintained, and each Feature is removed from the queue upon successful Response. This method is not recommended for higher velocity data feeds.

### `void send(Request, StreamObserver)`

The Asynchronous / non-Blocking request `send(Request, StreamObserver)` will send a single Request object, containing one or more Features, and return immediately. The passed in StreamObserver object can be used to process the Response. A single Response will be returned by the server for every Request. Use this method when a response from the server is desired for every request, but processing the response is not required before the next Request is sent. This method is not recommended for higher velocity data feeds.

### `StreamObserver stream(StreamObserver)`

The Asynchronous / non-Blocking request `stream(StreamObserver)` will open a stream to send multiple Request objects, each containing one or more Features. Sending a request will not result in a Response from the server. The returned request StreamObserver can be used to send individual requests using the `onNext(Request)` method and close the stream using the `onCompleted()` method. The `stream()` method is recommended for medium velocity data feeds.

The passed in response StreamObserver object can be used to monitor for errors in the underlying channel, process the final Response object, and verify the stream is closed. A single Response will be returned by the server when the stream is closed (no other response will be sent). Use this method when a response from the server is not needed for every request.

When using the `stream()` method, care should be taken on the client side to allow the server some time to process Requests. At the moment, there is no strong guideline in predicting the server's capacity or overload thresholds.

## Flow Control

gRPC supports the concept of flow control. This technical capability adds an additional communication channel between the client and server allowing the two applications to report their capacity to process data. If the Velocity gRPC server is overloaded, it will communicate this to the client, allowing it the opportunity to automatically throttle the event velocity.

Implementing flow control will allow the client application to obtain the maximum message velocity possible while avoiding overloading (and potentially crashing) the Velocity gRPC feed server. Flow control implementation is language specific (and may not be available in all client languages). In general,

the stub will offer a request stream observer with a `ready()` function that can be used to send requests from a queue. The server will trigger the `ready()` method on the client when it can process more data.

## Feature Wrapper

When sending data from the client to the server, the structure of the data is critical as it must adhere to the sample data structure provided to the Velocity gRPC feed. Because of this, the client message should be wrapped by an adapter that will properly add the data fields to the gRPC Feature object. Each field is packed into the Feature as an Any value object. Each field type has a translator available to convert standard data types (string, short, long, Boolean, and more) into the gRPC data value types. The gRPC data value types are then passed to the Any object to be packed into the feature.

**EXAMPLE:** A Java example of packing data into a Feature is highlighted below:

```
Feature.Builder featureBuilder = Feature.newBuilder();
featureBuilder.addAttributes(Any.pack(Int64Value.of(gpsRadio.ReportID)));
featureBuilder.addAttributes(Any.pack(StringValue.of(gpsRadio.DeviceID)));
featureBuilder.addAttributes(Any.pack(DoubleValue.of(gpsRadio.Latitude)));
featureBuilder.addAttributes(Any.pack(BoolValue.of(gpsRadio.IsEmergency)));
return featureBuilder.build();
```

## Error handling

The response codes listed below are generally used by most languages that implement gRPC. More details on the response codes and error handling can be found on the gRPC.io site:

<https://grpc.io/docs/guides/error/>

### gRPC Response Codes

Code	Number	Description
OK	0	Not an error; returned on success.
CANCELLED	1	The operation was cancelled, typically by the caller.
UNKNOWN	2	Unknown error. For example, this error may be returned when a <code>Status</code> value received from another address space belongs to an error space that is not known in this address space. Also, errors raised by APIs that do not return enough error information may be converted to this error.
INVALID_ARGUMENT	3	The client specified an invalid argument. Note that this differs from <code>FAILED_PRECONDITION</code> . <code>INVALID_ARGUMENT</code> indicates arguments that are problematic regardless of the state of the system (for example, a malformed file name).
DEADLINE_EXCEEDED	4	The deadline expired before the operation could complete. For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long.

NOT_FOUND	5	Some requested entity (for example, file or directory) was not found. Note to server developers: if a request is denied for an entire class of users such as gradual feature rollout or undocumented allowlist, NOT_FOUND may be used. If a request is denied for some users within a class of users such as user-based access control, PERMISSION_DENIED must be used.
ALREADY_EXISTS	6	The entity that a client attempted to create (for example, file or directory) already exists.
PERMISSION_DENIED	7	The caller does not have permission to execute the specified operation. PERMISSION_DENIED must not be used for rejections caused by exhausting some resource (use RESOURCE_EXHAUSTED instead for those errors). PERMISSION_DENIED must not be used if the caller can not be identified (use UNAUTHENTICATED instead for those errors). This error code does not imply the request is valid or the requested entity exists or satisfies other pre-conditions.
RESOURCE_EXHAUSTED	8	Some resource has been exhausted, perhaps a per-user quota, or perhaps the entire file system is out of space.
FAILED_PRECONDITION	9	The operation was rejected because the system is not in a state required for the operation's execution. For example, the directory to be deleted is non-empty, an rmdir operation is applied to a non-directory, and more. Service implementors can use the following guidelines to decide between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE: (a) Use UNAVAILABLE if the client can retry just the failing call. (b) Use ABORTED if the client should retry at a higher level (for example, when a client-specified test-and-set fails, indicating the client should restart a read-modify-write sequence). (c) Use FAILED_PRECONDITION if the client should not retry until the system state has been explicitly fixed. For example, if an rmdir fails because the directory is non-empty, FAILED_PRECONDITION should be returned since the client should not retry unless the files are deleted from the directory.
ABORTED	10	The operation was aborted, typically due to a concurrency issue such as a sequencer check failure or transaction abort. See the guidelines above

		for deciding between <code>FAILED_PRECONDITION</code> , <code>ABORTED</code> , and <code>UNAVAILABLE</code> .
<code>OUT_OF_RANGE</code>	11	The operation was attempted past the valid range. For example, seeking or reading past end-of-file. Unlike <code>INVALID_ARGUMENT</code> , this error indicates a problem that may be fixed if the system state changes. For example, a 32-bit file system will generate <code>INVALID_ARGUMENT</code> if asked to read at an offset that is not in the range <code>[0, 2^32-1]</code> , but it will generate <code>OUT_OF_RANGE</code> if asked to read from an offset past the current file size. There is a fair bit of overlap between <code>FAILED_PRECONDITION</code> and <code>OUT_OF_RANGE</code> . It is recommended to use <code>OUT_OF_RANGE</code> (the more specific error) when it applies so that callers who are iterating through a space can easily look for an <code>OUT_OF_RANGE</code> error to detect when they are done.
<code>UNIMPLEMENTED</code>	12	The operation is not implemented or is not supported/enabled in this service.
<code>INTERNAL</code>	13	Internal errors. This means that some invariants expected by the underlying system have been broken. This error code is reserved for serious errors.
<code>UNAVAILABLE</code>	14	The service is currently unavailable. This is most likely a transient condition, which can be corrected by retrying with a backoff. Note that it is not always safe to retry non-idempotent operations.
<code>DATA_LOSS</code>	15	Unrecoverable data loss or corruption.
<code>UNAUTHENTICATED</code>	16	The request does not have valid authentication credentials for the operation.

## Appendix

### ArcGIS Velocity Swagger API

Access the ArcGIS Velocity Swagger API for an organization at the following URL pattern:

```
https://velocity.arcgis.com/<clusterID>/<orgID>/iot/api/swagger.ht  
ml
```

The **ClusterID** can be found in the REST API URL of any of the feeds as the first GUID in the URL string.

The **orgID** can be found in the REST API URL of any of the feeds as the second GUID in the URL string.

The **ClusterID** and the **orgID** may be identical to one another or different.