# KIP-502: Connect Task Method Signature to be More Specific for Developers

## Status

**Current state**: Under Discussion

**Discussion thread**: here

**JIRA**: here

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

## Motivation

*TLDR: When a developer implements `SinkTask.put(...)`, it's a much better experience to be guaranteed an `List<SinkRecord>` than the current `Collection<SinkRecord>`, because the connector developer will have access to the full set of List methods and not just limit set of Collection methods.*

When a developer writes a new Sink Connector, the key class to implement is `SinkTask`, and the key method `SinkTask.put(...)` does the work of processing records from Kafka. So it's critical that developers have the right method signature here to work with. Its current abstract class specifies:

```
    public abstract void put(Collection<SinkRecord> records);
```

Usually, the implementer will use a for-each loop to iterate through the records one by one and act on them.

There are two shortcomings with this signature:

1. The Connector implementation is not provide with any **ordering guarantees** of Records.
2. The proposed change will give **flexibility to task implementation** by giving it more knowledge, over the method caller who prefers to givew few guarantees.

### Ordering Guarantees

Kafka gives ordering guarantees within a Topic-Partition: Records will have strictly increasing offsets as they are consumed. The current SinkTask forfeits any guarantee when it passes the records to the Task via the `put(Collection<SinkRecrod>)` method. From the Task's perspective, it is getting a list of maybe-ordered records. If a task wants to put records in S3 with a file for each topic-partition, a task **must** sort the collection of records before putting them into S3, since it is not guaranteed any ordering from the caller. We happen to know that the only caller of SinkTask.put(...) (which happens to be WorkerSinkTask) does indeed call put(...) with an List that is sorted in the order the consumer consumes. But why is this agreement tacit and not explicit? It would harden the API to explicitly require the transfer of records to be in a datatype that maintains ordering.

### Flexibility for Implementers

Generally, whenever we choose a signature for a function, we balance a tension between the flexibility of the callers of the method and the implementations of the signature. The caller will always want to have the most general interface as possible: This will let the caller pass in a variety of objects as a valid argument. Conversely, the implementer will want to know specifically what kind of object it needs to be able to handle. It will "unlock" more methods if it is guaranteed a more specific type of object, and can rely on more assumptions about specific types that it is forbidden from doing in the general case (e.g. constant time lookup by index is valid to assume in an List but not a Collection in general).

Concretely, as an implementer of a connector's SinkTask.put(...) method, I want to do things like `records.get(int i)`. I can only do this if the framework gives a more specific implementation of Collection like List.

**The purpose of this KIP is to reconsider the interface's use of "Collection<SinkRecord>", in favor of the more specific datatype "List<SinkRecord>".**

## Public Interfaces

Add to the "SinkTask" abstract class a method:

```
    public abstract void put(List<SinkRecord> records);
```

## Proposed Changes

Add to the "SinkTask" abstract class a method:

```
    public abstract void put(List<SinkRecord> records);
```

Implement the existing abstract method to call the new method in a backwards compat way:

```
    public void put(Collection<SinkRecord> records) {
        ArrayList<SinkRecord> recordsList = records instanceof ArrayList
            ? (ArrayList<SinkRecord>) records : new ArrayList<>(records);
        put(recordsList);
    }
```

## Compatibility, Deprecation, and Migration Plan

It is critical that connect frameworks before and after the change are compatible with connectors implemented with either signature. It is less critical that developers of connectors will be able to pick which to implement if they upgrade their artifacts to new AK version as a dep.

The framework will not change, just the Abstract class. The existing (unchanged) call within WorkerSinkTask of `task.put(new List<>(messageBatch));` will successfully call the right function whether the connector has implemented the new or old signature.

Connector developers will need to change their SinkTask implementations if they update their artifacts, but this is acceptable since they'll find out at compile-time during their development process, not at runtime on production.

There is only one caller of this method, `WorkerSinkTask`, which always passes an ArrayList as the Collection.

## Rejected Alternatives

### ArrayList

Considered using ArrayList since you'd get guarantees about runtime of the indexing, but decided it is a bad tradeoff to get this guarantee at the expense