mojaloop

Mojaloop Versioning

PI 10 - April 2010

Lewis Daly, Matt de Haast, Samuel Kummary

Agenda

1. Goal and Background
2. Versioning Proposal
3. Versioning Support + Implementation Proposal
4. Next Steps

Goal

Propose a standard for a new 'Mojaloop Version', which embodies:

1. **API Versions**
   1.1. FSPIOP API
   1.2. Hub Operations / Admin API
   1.3. Settlement API
2. **Helm**: Individual Service Versions, Monitoring Component Versions
3. **Internal Schema Versions**: Database Schema & Internal Messaging Data Model

Motivation

Need for versioning

1. Settlement v2
2. Cross currency / network
3. PISP related changes
4. Other Updates, features, enhancements
5. Newer releases of the APIs – FSPIOP, Settlement, Operations / Admin

Proposal

“Mojaloop Version”

- Format: **x.y**.z: MAJOR.MINOR.HOTFIX

  (HOTFIX: for internal use only, not broadcasted publicly)

Mojaloop	**x.y**		
	Owner Maintainer	Format	Meaning
APIs			
- FSPIOP	CCB	x.y	Major.Minor
- Settlement	CCB	x.y	Major.Minor
- Admin/Operations	CCB	x.y	Major.Minor
Helm	Design Authority	x.y.z	PI.Sprint.Iteration
Internal Schemas			
- DB Schema	Design Authority	x.y	Major.Minor
- Internal Messaging	Design Authority	x.y	Major.Minor

Read the full proposal here.

Example: Mojaloop v1.0

Mojaloop	**1.0**	
	Owner/Maintainer	Version
APIs		
- FSPIOP	CCB	1.0
- Settlement	CCB	1.1
- Admin/Operations	CCB	1.0
Helm	Design Authority	9.2.0
Internal Schemas		
- DB Schema	Design Authority	1.0
- Internal Messaging	Design Authority	1.0

Advantages

1. **Simplicity**
   a. A given version say - Mojaloop v1.0 refers to three APIs, along with the Helm version that is a bundle of the individual services which are compatible with each other and can be deployed together.
   b. Schema versions for the DB and Internal messaging to communicate whether any changes have been made to these or not since the previously released version.

2. **One size fits all:**
   a. It caters for everyone who may be interested in differing levels of details, whether high level or detailed.
   b. Because of the nature of the major and minor versions, it should be easy for **users** and **adopters** and **developers** to understand compatibility issues

Compatibility

As described in section 3.3 of the API Definition v1.0, whether or not a version is backwards compatible is indicated by the **Major** version. All versions with the same major version must be compatible while those having different major versions, will most likely not be compatible.

Given 3 Participants:

Participant	**Supported Versions**
DFSP A	`1.0, 1.1`
DFSP B	`1.1, 2.0`
DFSP C	`2.0`

Compatibility

Participant	Supported Versions
DFSP A	1.0
DFSP B	1.1, 2.0
DFSP C	2.0

```
A@v1.0 <---> B@v1.1: OK     - MINOR versions are compatible
B@v2.0 <---> C@v2.0: OK     - MAJOR versions match

A@v1.0 <---> B@v2.0: NOT OK - MAJOR versions are incompatible
A@v1.0 <---> C@v2.0: NOT OK - MAJOR versions are incompatible
B@v1.1 <---> C@v2.0: NOT OK - MAJOR versions are incompatible
```

API Versioning* [ML FSPIOP API]

1. **Major minor versions – Minor version change guideline**
   a. Optional input parameters such as query strings added in a request
   b. Optional parameters added in a request or a callback
   c. Error codes added
2. **Major version change guideline**
   a. Mandatory parameters removed or added to a request or callback
   b. Optional parameters changed to mandatory in a request or callback
   c. Parameters renamed
   d. Data types changed
   e. Business logic of API resource or connected services changed
   f. API resource/service URIs changed

Client Side Negotiation [ref: api spec section 3.3.4]

1. Client specifies in the `Accept` Header:

```
POST /service HTTP/1.1
Accept: application/vnd.interoperability.{resource}+json;version=1,
application/vnd.interoperability.{resource}+json

{
    ...
}
```

2. Server responds with an acceptable version

```
Content-Type: application/vnd.interoperability.{resource}+json;version=1.0
{
    ...
}
```

Or an error callback

```
{
  "errorInformation":{
   "errorCode": "3001",
   "errorDescription": "The Client requested an unsupported version, see extension list for supported version(s).",
   "extensionList": [ { "key": "1", "value": "0" }, { "key": "2", "value": "1" }, { "key": "4", "value": "2" }
   ]
  }
 }
```

Implementation - Considerations

1. Standardize branching, tagging mechanism on GitHub
2. Version negotiation - to be added
3. Handling major, minor versions in the implementation – Guidelines for schemes
4. Supported versions in implementation - latest versions
    a. Scheme Choices
    b. Guidance for Schemes
5. Timelines for support

A Breaking Change

Let's take an arbitrary breaking change to the Mojaloop Specification, and list through a number of strategies we can use to deal with such a change.

For example, a current proposal is to include the quoteId in the `POST /transfers` request from DFSPA to the Switch:

```
POST /transfers HTTP/1.1
Accept: application/vnd.interoperability.transfers+json;version=2
Date: Tue, 15 Nov 2020 08:12:31 GMT
... other headers...
Body:
{
  "transferId": "1234",
  "quoteId": "9876", <--- This is a new required field
  "payeeFsp": "dfspb",
  "payerFsp": "dfspa",
  "amount": "100.00",
  "ilpPacket": "XXXXX",
  "condition": "XXXXX",
  "expiration": "2020-05-24T08:38:08.699-04:00"
}
```

Implementation - Approaches

1. **“Stop the World”**

2. **“Worry About it in the Infrastructure”**

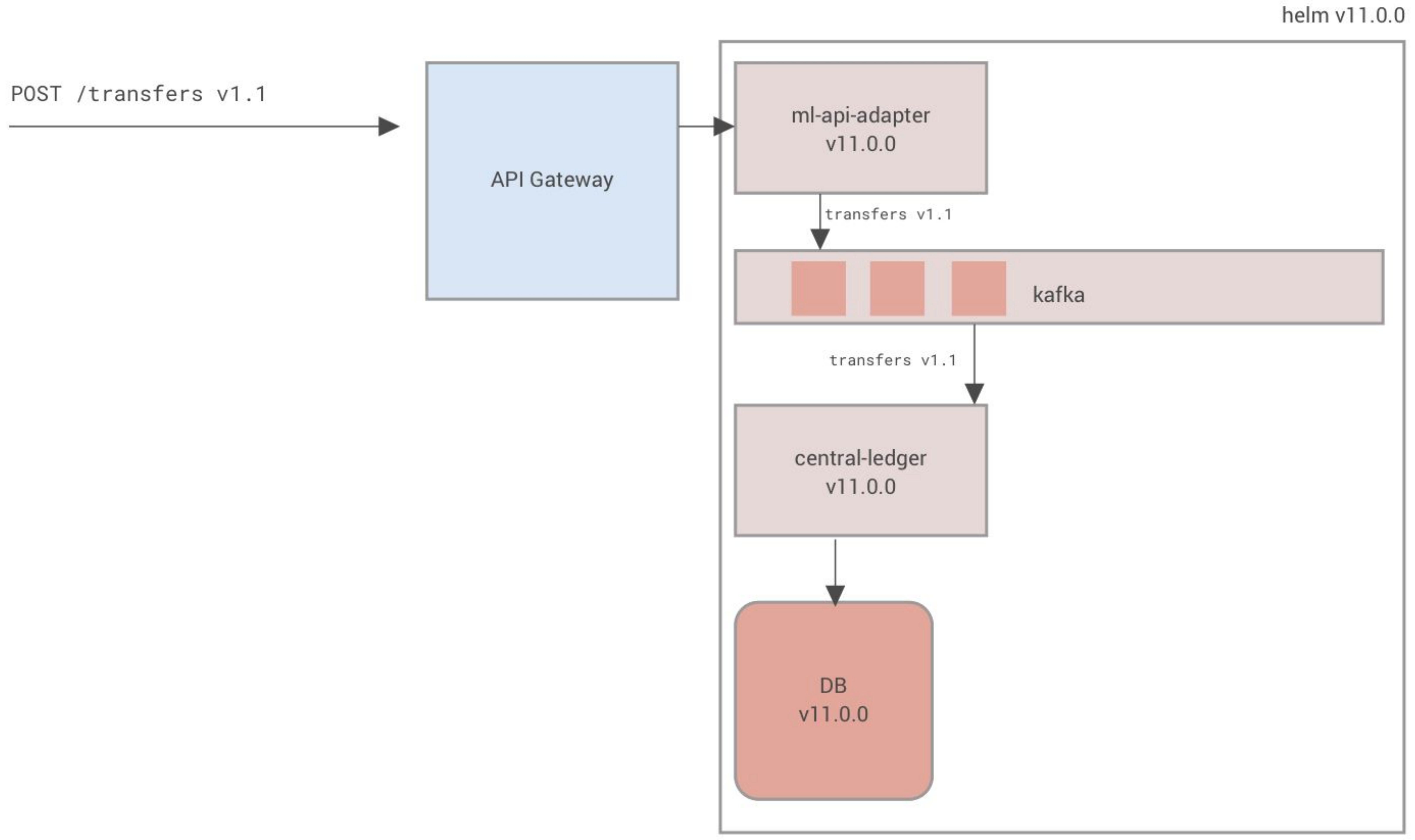
   (Run multiple versions of the same service internally)

3. **“Worry About it in the Code”**

   (The latest version of each service is *always* backwards compatible)

Let’s work through all the examples first, then we can have an open discussion.

Simplified POST `/transfers`



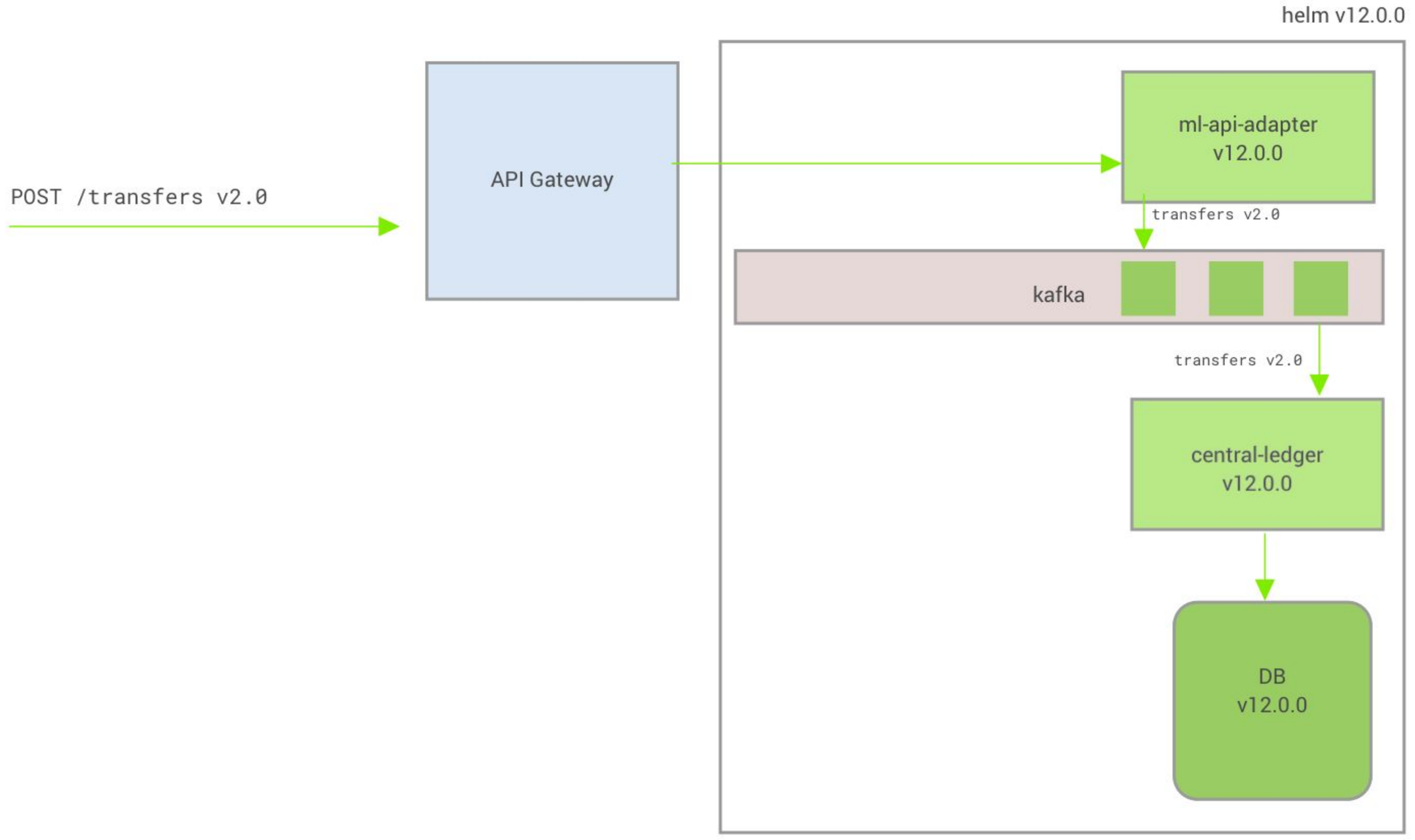
1. Stop the World

Simply take down the entire deployment, run any schema migrations, and start it up again

This is what we have (by default) today.

Steps:

1. At the API Gateway level, stop any incoming quote requests, and wait until the pending transfers are complete or timed out. This is essentially draining the messages inside of the kafka streams
2. Once again at the API Gateway, kill all traffic to the switch
3. Assuming database persistence, run `helm upgrade` to update the helm charts from v11.0.0 to v12.0.0
4. Behind the scenes, when the new helm containers start up, the database migrations will be run
5. Once all services are healthy, re-enable all traffic and we are good to go.
    - Note that since we did nothing around supporting multiple FSPIOP API versions, *by default* the upgraded switch will *only* support POST /transfers requests at version 2.


POST /transfers v2.0
API Gateway
helm v12.0.0
ml-api-adapter
v12.0.0
transfers v2.0
kafka
transfers v2.0
central-ledger
v12.0.0
DB
v12.0.0

2. Worry about it in the Infrastructure

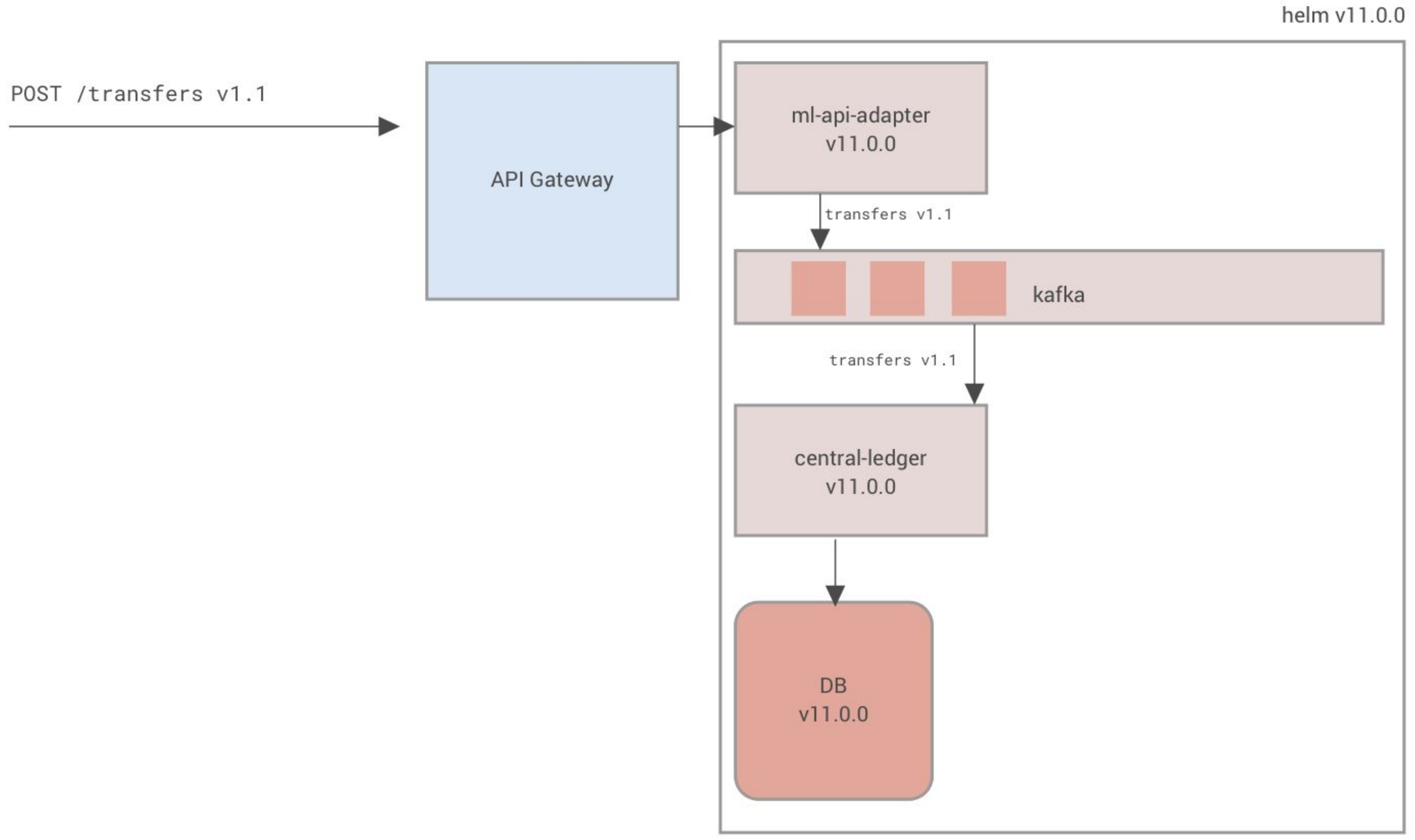
1. Use kubernetes and helm to help us manage and run multiple 'internal' versions of our microservices simultaneously.
2. Partition kafka based on message *versions* to ensure the right services pick up the right messages.

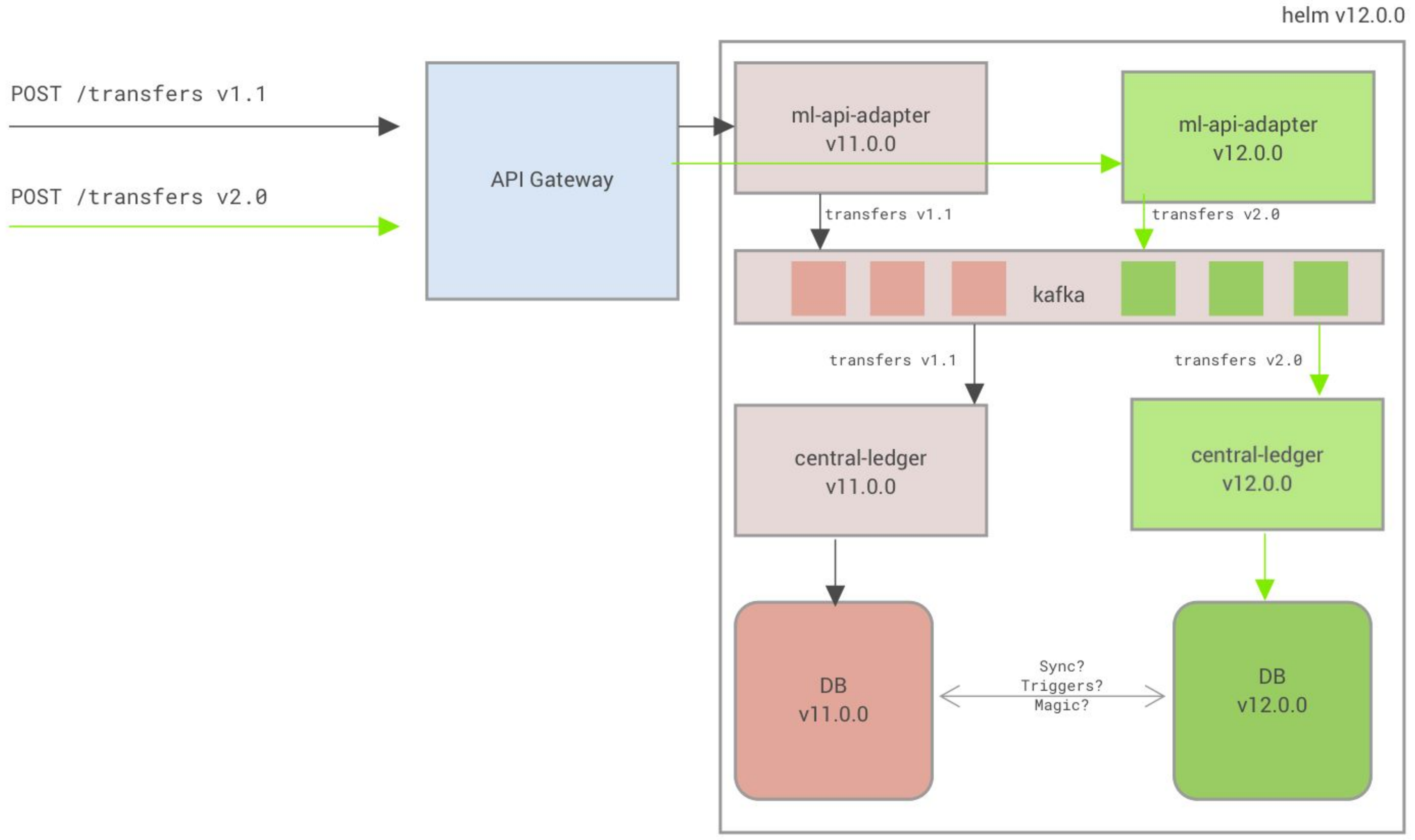
Given the following

- `central-ledger:v11.0.0`, which can understand POST /transfer requests of v1.1, without the additional quoteId field.
- `central-ledger:v12.0.0`, which can understand POST /transfer requests of v2.0, *with* the additional quoteId field.

Inside of a *single* helm deployment, we can run both 2 versions of central-ledger, v11.0.0 and v12.0.0 at the same time.

Assuming some routing magic that can send the right version of requests to the right service, this gives us the ability to understand POST /transfer requests of both versions.


helm v11.0.0
POST /transfers v1.1
API Gateway
ml-api-adapter
v11.0.0
transfers v1.1
kafka
transfers v1.1
central-ledger
v11.0.0
DB
v11.0.0


helm v12.0.0
POST /transfers v1.1
POST /transfers v2.0
API Gateway
ml-api-adapter v11.0.0
ml-api-adapter v12.0.0
transfers v1.1
transfers v2.0
kafka
transfers v1.1
transfers v2.0
central-ledger v11.0.0
central-ledger v12.0.0
DB v11.0.0
Sync? Triggers? Magic?
DB v12.0.0

2. Worry about it in the Infrastructure

Steps:

1. Author a new Helm version v12.0.0, which:
   - spins up a new database alongside the existing database, instead of replacing it [or multiple database *views*?]
   - adds new services (e.g. central-ledger:v12.0.0), instead of replacing them
2. While still not allowing v2 requests at the API Gateway, `helm upgrade` to spin up new infrastructure alongside the existing infrastructure
3. Execute database schema migration and views/table sync etc.
4. Once all services are healthy, open up traffic to v2.0, and we are good to go

Other Thoughts:

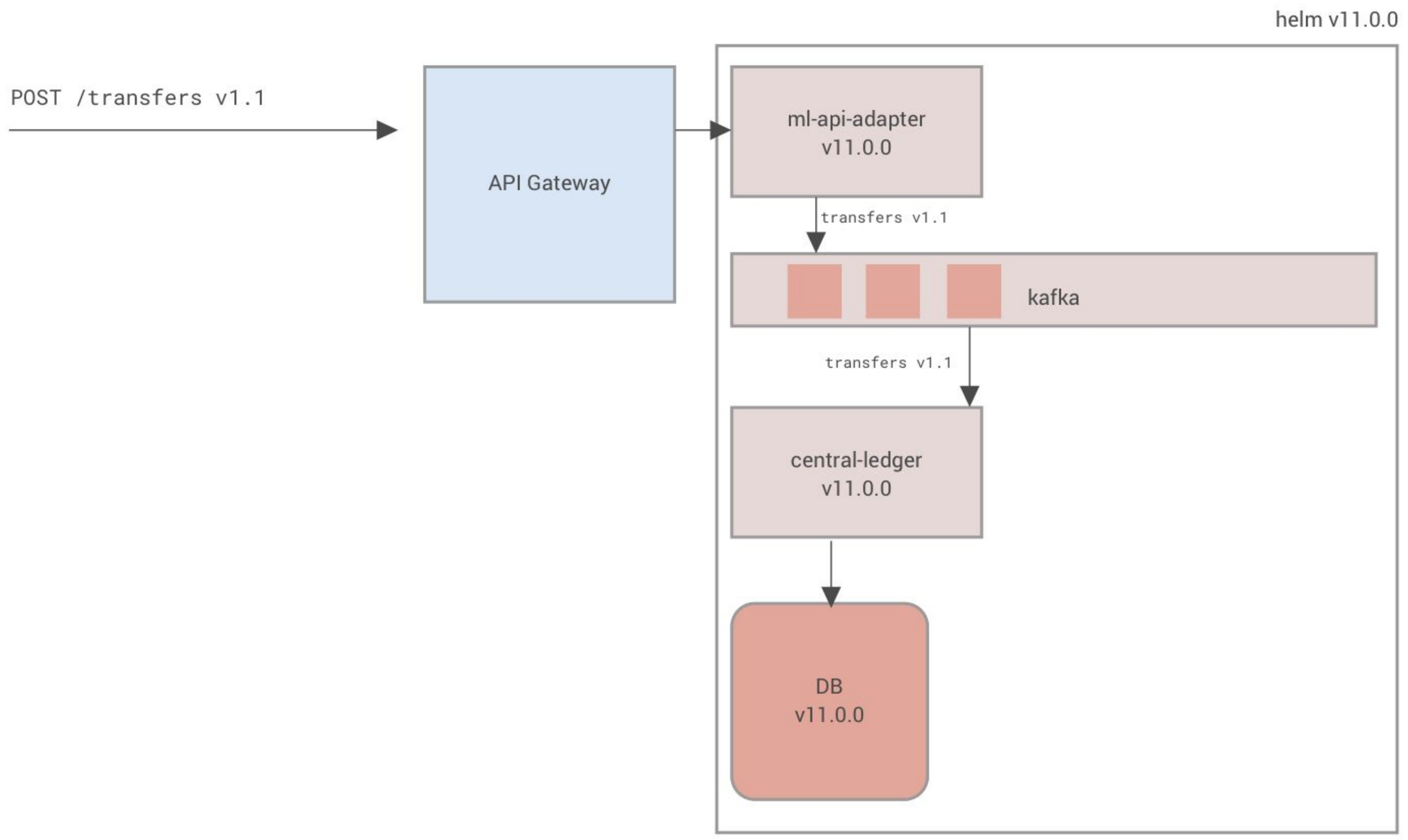
- This method supports multiple API versions at the same time, but
  - Requires a lot of infrastructure heavy lifting for such a small change. (What about the next change? Do we need to run another copy of almost all the services?)
  - Does not allow DFSPs at different MAJOR versions to interoperate (which it may not need to).
  - It could impact Kafka performance by partitioning topics per message version

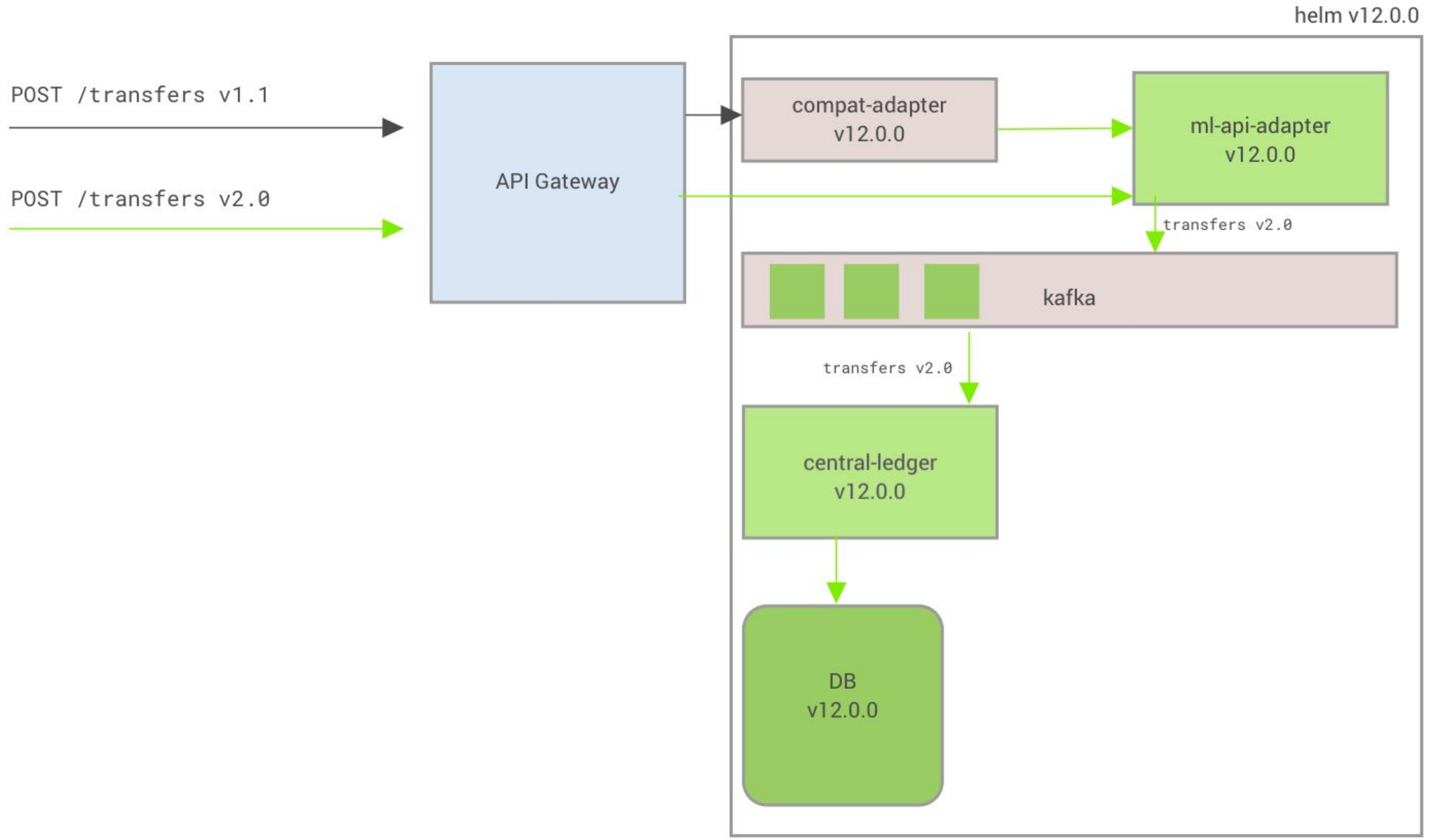
3. Worry about it in the Code

- In this approach (as the title implies), worry about such changes in the code.
- Don't rely upon helm and kubernetes to run multiple versions of a given service at the same time
- Our services *always talk the latest API*, and in order to talk the older versions of the API, we *adapt messages* forwards and backwards.

The advantage with this approach is that we can always run and maintain the *latest* version of our services, and minimize the need to touch legacy code.

Once again, let's take a look at our simplified transfer request flow:


helm v11.0.0
POST /transfers v1.1
API Gateway
ml-api-adapter v11.0.0
transfers v1.1
kafka
transfers v1.1
central-ledger v11.0.0
DB v11.0.0


POST /transfers v1.1
POST /transfers v2.0
API Gateway
helm v12.0.0
compat-adapter v12.0.0
ml-api-adapter v12.0.0
transfers v2.0
kafka
transfers v2.0
central-ledger v12.0.0
DB v12.0.0

3. Worry about it in the Code

For this approach, we define a new service called compat-adapter, which is responsible for upgrading incoming messages and downgrading outgoing messages to the correct API versions.

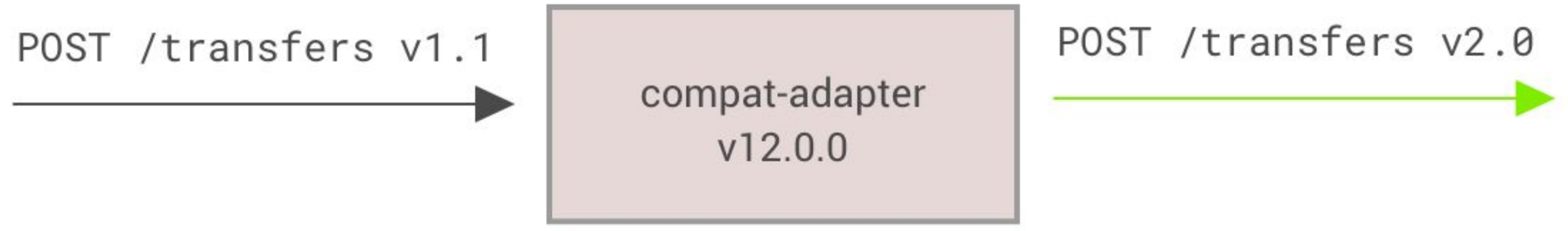
By default, once we deploy helm v12.0.0, the mojaloop switch talks transfers v2.0. In order to understand and responds to transfers v1.1 messages, the compat-adapter does the work in upgrading the incoming messages, and downgrading the outgoing messages to the relevant API versions.

3. Worry about it in the Code

For this approach, we define a new service called compat-adapter, which is responsible for upgrading incoming messages and downgrading outgoing messages to the correct API versions.

By default, once we deploy helm v12.0.0, the mojaloop switch talks transfers v2.0. In order to understand and responds to transfers v1.1 messages, the compat-adapter does the work in upgrading the incoming messages, and downgrading the outgoing messages to the relevant API versions.

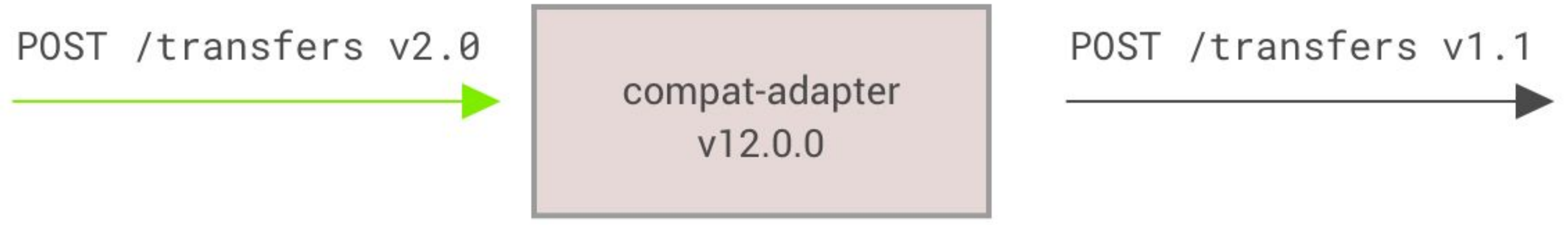
Upgrading Incoming Messages



```
Accept: application/vnd.interoperability
          .transfers+json;version=1.1
Body:
{
  "transferId": "1234",
  "payeeFsp": "dfspb",
  "payerFsp": "dfspa",
  ...
}
```

```
Accept: application/vnd.interoperability
          .transfers+json;version=2.0
Body:
{
  "transferId": "1234",
  "quoteId": null,
  "payeeFsp": "dfspb",
  "payerFsp": "dfspa",
  ...
}
```

Downgrading Outgoing Messages



```
Accept: application/vnd.interoperability
         .transfers+json;version=2.0
Body:
{
  "transferId": "1234",
  "quoteId": "5678",
  "payeeFsp": "dfspb",
  "payerFsp": "dfspa",
  ...
}
```

```
Accept: application/vnd.interoperability
         .transfers+json;version=1.1
Body:
{
  "transferId": "1234",
  "payeeFsp": "dfspb",
  "payerFsp": "dfspa",
  ...
}
```

mojaloop

3. Worry about it in the Code

Steps

1. Author a new Helm v11.1.0, which:
   - Applies any necessary database and message schema changes for upcoming version v12.0.0
2. Deploy Helm v11.1.0 with helm upgrade. Database migrations will be applied live
3. Update internal services to be able to handle the new transfer.quoteId, as an optional field, and implement business logic for handling cases without it
4. Bundle these internal service changes in Helm v11.2.0
5. Deploy Helm v11.2.0 with helm upgrade. Internal services will now be able to handle the transfer.quoteId field, but since all messages are still v1.1, nothing has changed.
6. Build/update compat-adapter service to handle upgrading and downgrading requests and responses to and from v1.1 and v2.0
7. Bundle the compat-adapter in with Helm v12.0.0
8. Deploy helm v12.0.0, which will start the compat-adapter service
9. Once all services are healthy, open up traffic to v2.0 requests at the API Gateway, and we are good to go

3. Worry about it in the Code

Caveats

1. There are no longer any guarantees about the internal messaging and schema for the `transfers.quoteId` field.
    - This means our internal services need to more heavy lifting to handle this, and we need to make some decisions about what to do to handle null values for `quoteId`

Next Steps

- **Further discussion about upgrade strategies**
  - Workshop? Extended discussion with interested OSS Members?

- **Proof of concept for zero-downtime-deployments**
  - Choose one of the approaches, and implement a narrowly scoped POC
  - End-to-end tests to prove:
    - zero-downtime
    - support of multi-dfsps at different versions
  - Feedback any process changes that need to become a core part of the Mojaloop dev cycle