Building a Global API Banking Platform using Kafka, RPC and Aerospike

Matteo Pelati

SVP, Data and API Engineering

DBS Bank

Typical Digital Banking Implementation

API Gateway

Experience Layer

Source System

Typical Challenges

- **Scalability and performance**: source systems are based on legacy technologies which are very hard to scale
- **Hard to maintain**: different countries and products often use different source systems, requiring different implementations to serve the same functionality
- **Typically batch-oriented**: real-time processing of data is extremely painful. This makes it hard to compete with the real-time services offered by new virtual banks

We need a better API layer which is

- **Fast and Scalable:** the system must be able to scale horizontally and offer extremely low latency
- **Easily query-able:** it should be easy to consume any API across different organizations of the bank
- **Country and System agnostic:** API consumers should not worry about the systems they are talking with
- **Easy to extend:** different groups across the bank should be able to easily plug-in their API endpoints
- **Support both push and pull mechanisms:** consumers should be able to query data or subscribe to events
- **Open Source stack:** minimize dependencies from vendors
- **Centralized, but self-service:** every user in the bank must be able to independently develop and deploy their APIs as part of the API ecosystem

Eventing + shadow database

Source System generates events notifying any state change (CICS, CDC) and push them to Kafka

Events are “projected” to the shadow database to replicate the data

Shadow database contains an eventually consistent replica of the source system

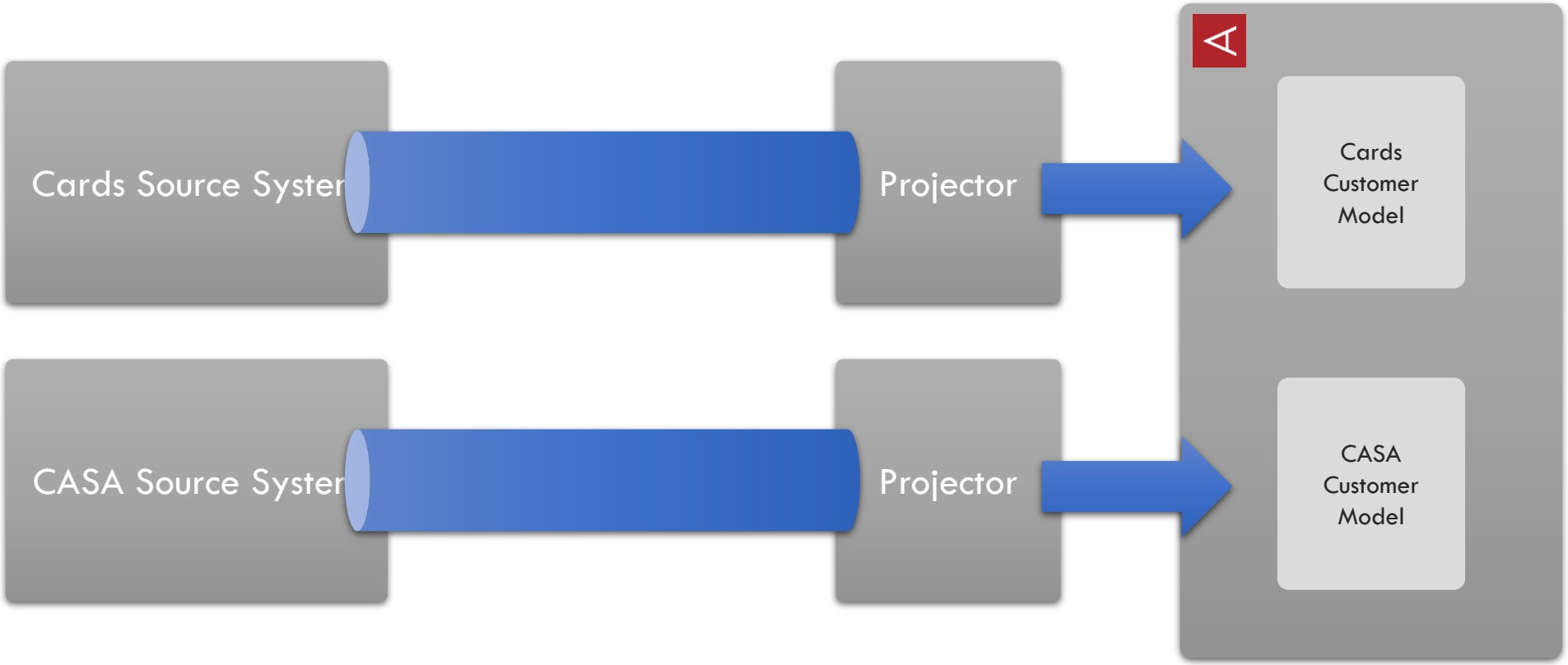
Database choice

	Aerospike	MongoDB	Redis	FDB
Fast key/value lookup	●	●	●	●
Open Source	●	●	●	●
Memory + flash storage	●	●	●	●
Shared-nothing architecture	●	●	●	●
Complex data structures	●	●	●	●
Secondary Indexes	●	●	●	●
Commercial Support	●	●	●	●

Based on the table above, we decided to adopt Aerospike as the database of choice for all services

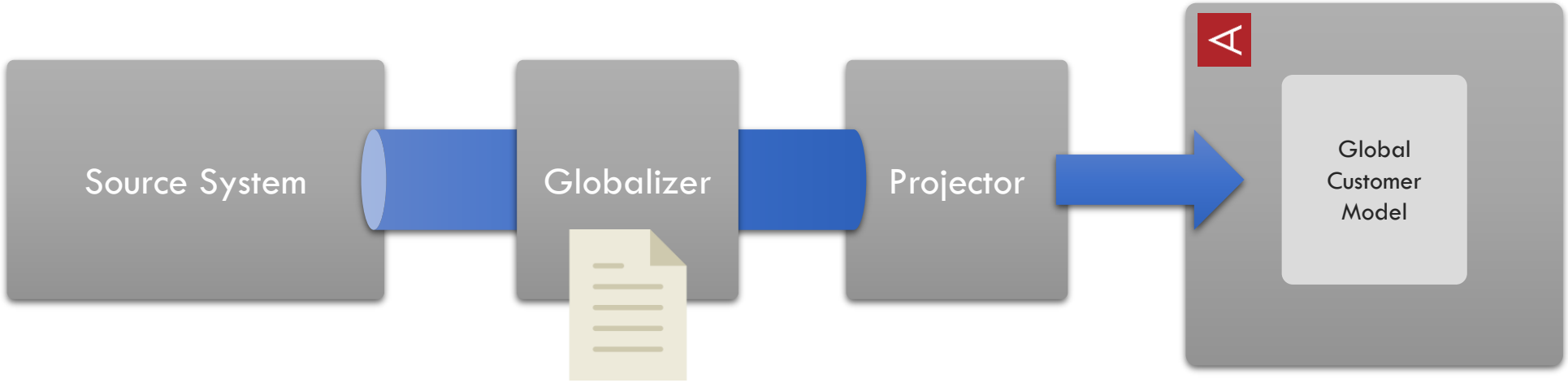
We needed a Global Model

The same business entity is very different from source system to source system. We need an efficient way to "globalize" the model stored in the database.



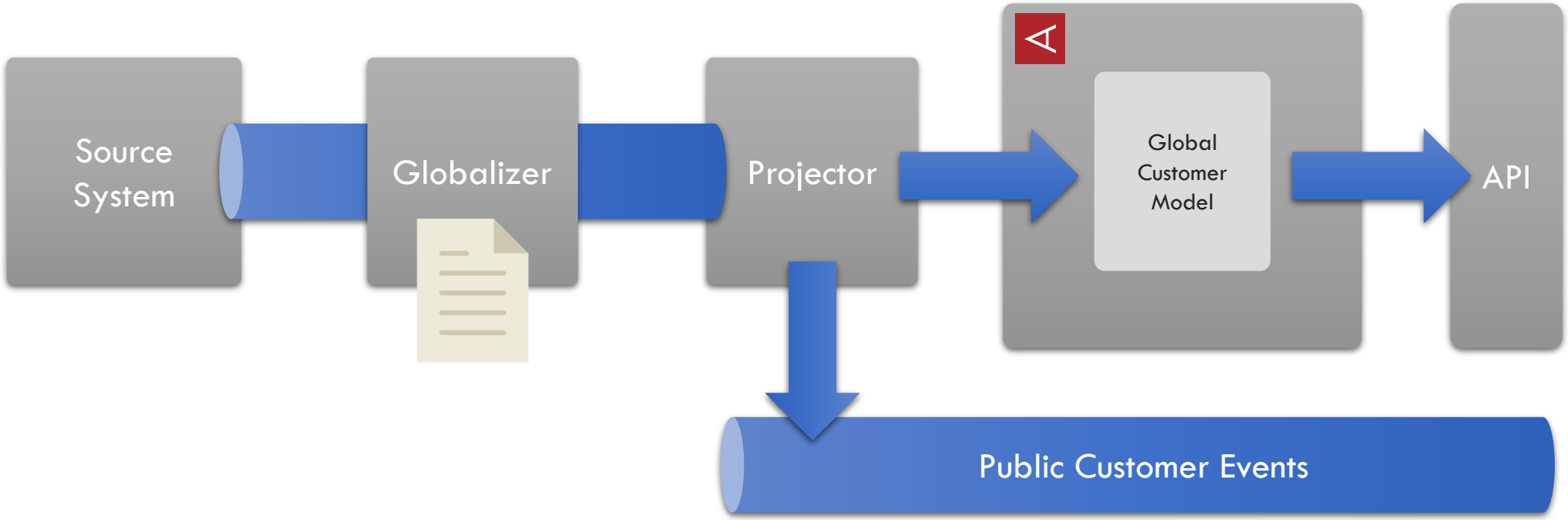
Introducing ”the Globalizer”

Every model gets translated into a Global model by a Globalizer component, which is fully configurable using a custom DSL. This way, domain developers can easily plug-in new source systems without requiring to write any code.



Serving the data

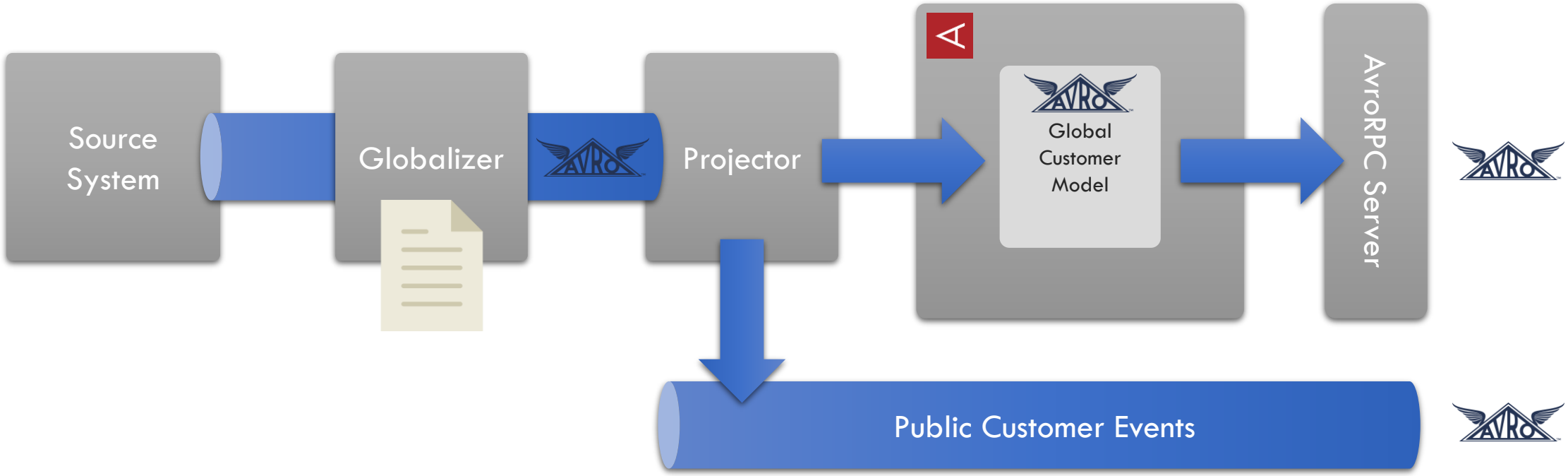
- **APIs** for accessing the current state of the database
- **Kafka topic** for listening to all updates of a specific domain



Data Format

- **De facto standard for streaming**: Avro is the format of choice of Apache Kafka
- **RPC Support**: Avro can seamlessly be used for implementing RPC services
- **Compact**: it's binary, thus it's fast
- **Evolutionary**: its supports very well schema evolution

Avro all the way



AVRO format is used from the Globalizer, all the way down to the RPC server, which is serving AVRO payloads to the client.

The Avro Record Format

How can we speed-up even further the responses? Just dump the AVRO payload into the database

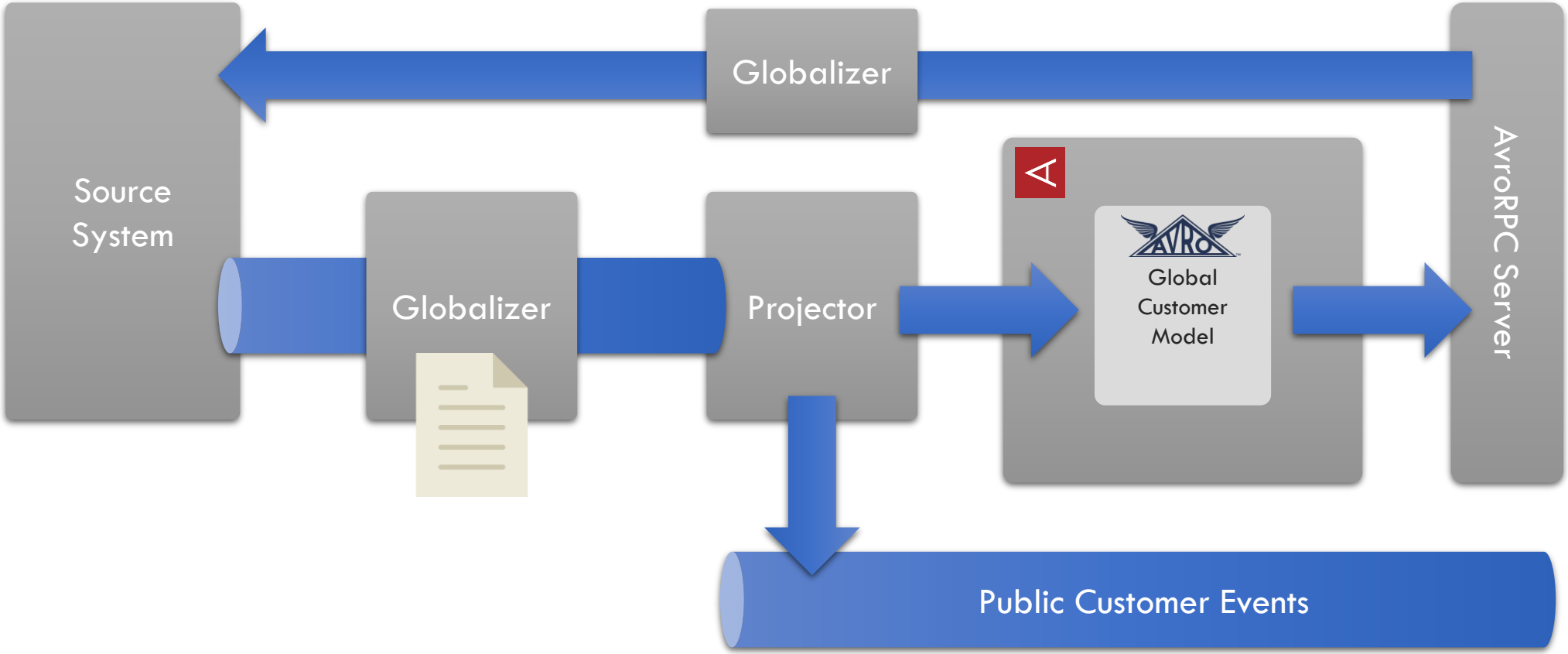
```
{
   "type" : "record",
   "namespace" : ”Customers",
   "name" : ”Customer",
   "fields" : [
      { "name" : ”CustomerId" , "type" : "string”, “pk”: true },
      { "name" : ”Country" , "type" : ”string”, “sk”: true }
   ]
}
```

Payload:	a002ef10cc76eb21964abbf3489
CustomerId	366635326
Country	SG

The Payload field is used to return the raw data to the client during an API call, while other fields are solely used for creating secondary indexes. This design was inspired by the Record Layer of FDB from Apple

We still have command APIs

Relying on the source system for transactional operations still requires us to interact with it. In such cases the RPC server directly invokes the source system APIs, but relying on eventing to keep the state up to date



API implementation as a service

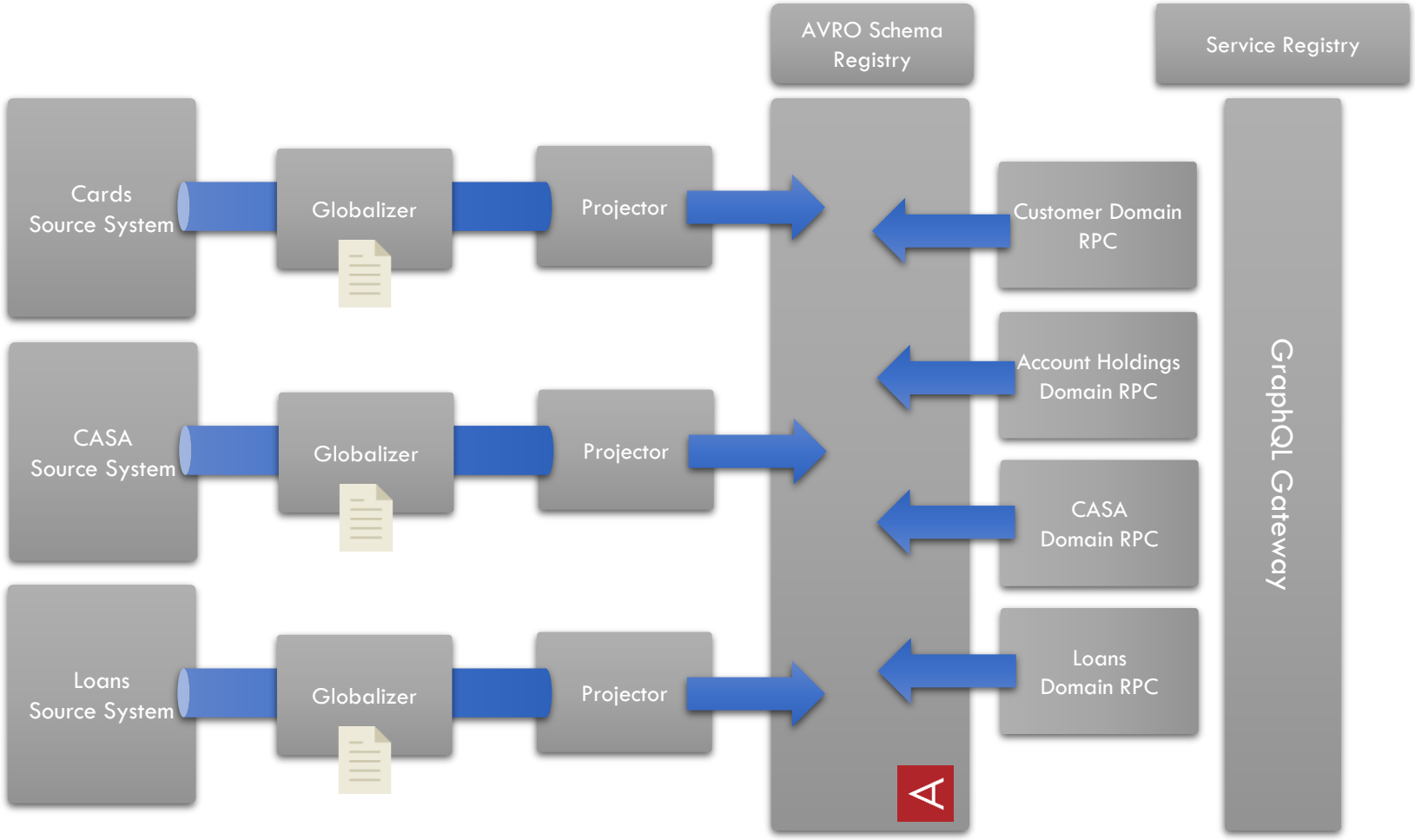
The goal of the platform is to enable domain developers to easily implement new APIs using a set of shared components. In particular:

- **Globalizer**: entirely configuration-driven. A DSL is used to configure mappings between legacy model and the new global Avro model
- **Projector**: entirely configuration driven. Used to project the data into the shadow database
- **RPC Server**: supports lambda-like plugins to allow developers to implement custom lookups

We still have challenges

- **Eventual consistency sometimes is a problem**: mobile apps are not designed based on eventual consistency principle. They expect after an action, data is updated immediately. That is not the case for an eventual consistent system.
- **The batch nature of core systems sometimes require the cache to be completely refreshed**: not always source systems can generate events in real time. Sometimes, after a batch operation, the cache needs to be refreshed in bulk

GraphQL: how everything sticks together




```
query {
   customer(id: 17268228) {
      city
   }
}
```

Thank You

Contact me at matteopelati@dbs.com if you'd like to work with us!!