Table of Contents

Distributing into Microservices (15 pages)

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> — Leslie Lamport

Thus far, we have a working application for LC application processing, which is bundled along with other components as a single package. Although, we have discussed the idea of subdomains and bounded contexts, the separation between these components is logical, rather than physical. Furthermore, we have focused primarily on the *LC Application Processing* aspect of the overall solution. In this chapter, we will look at extracting the LC Application Processing bounded context into components that are physically disparate, and hence enable us to deploy them independently of the rest of the solution. We will discuss various options available to us, the rationale for choosing a given option, along with the implications that we will need to be cognizant of.

Continuing our design journey

From a logical perspective, our realization of the Letter of Credit application looks like the visual depicted here:

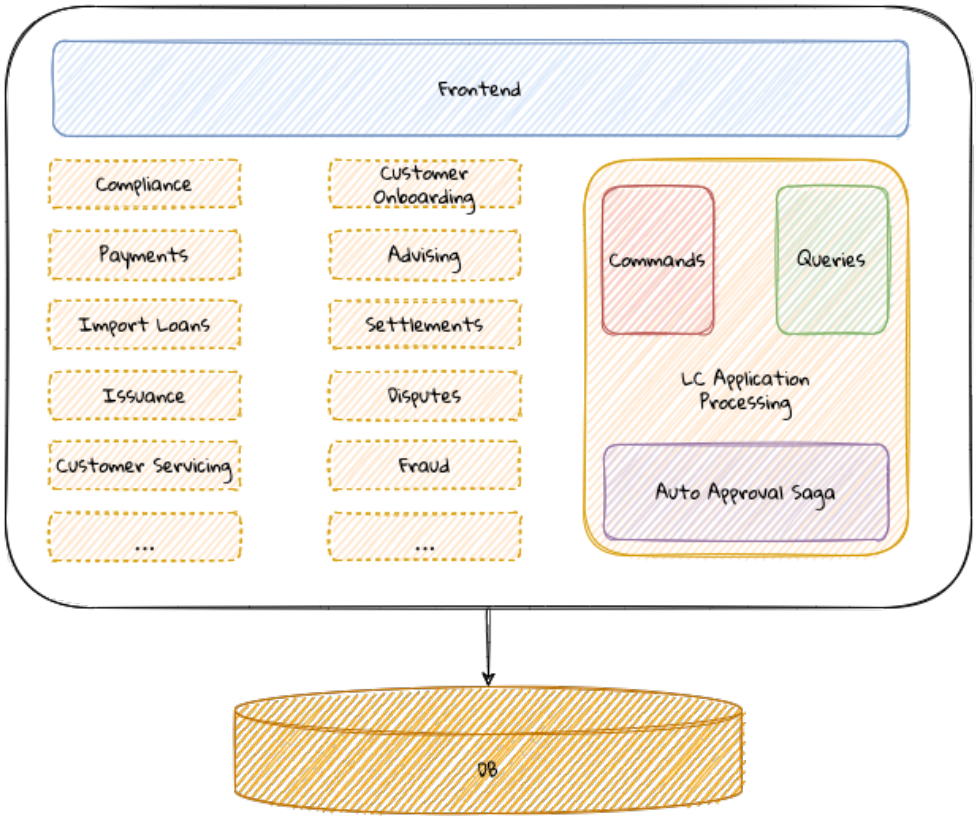


Figure 1. Current view of the LC application monolith

Although the *LC Application Processing* component is loosely coupled from the rest of the application, we are still required to coordinate with several other teams to realize business value. This may inhibit our ability to innovate at a pace faster than the slowest contributor in the ecosystem. This is because all teams need to be production ready before a deployment can happen. This can be further exacerbated by the fact that individual teams may be at different levels of engineering maturity. Let's look at some options on how we can achieve a level of independence from the rest of the ecosystem by physically decomposing our components into distinctly deployable artifacts.

Decomposing our monolith

First and foremost, the LC Application Processing component exposes only in-process APIs when other components interact with it. This includes interactions with:

1. Frontend
2. Published/consumed events
3. Database

To extract LC application processing functionality out into its own independently deployable component, remotely invokable interfaces will have to be supported instead of the in-process ones we have currently. Let's examine remote API options for each:

Changes for frontend interactions

Currently, the JavaFX frontend interacts with the rest of the application by making request-

response style in-process method calls (`CommandGateway` for commands and `QueryGateway` for queries) as shown here:

```
@Service
public class BackendService {

    private final QueryGateway queryGateway;
    private final CommandGateway commandGateway;

    public BackendService(QueryGateway queryGateway,
                          CommandGateway gateway) {
        this.queryGateway = queryGateway;
        this.commandGateway = gateway;
    }

    public LCApplicationId startNewLC(ApplicantId applicantId, String clientReference)
{
        return commandGateway.sendAndWait(
                startApplication(applicantId, clientReference));
    }

    public List<LCView> findMyDraftLCs(ApplicantId applicantId) {
        return queryGateway.query(
                new MyDraftLCsQuery(applicantId),
                        ResponseTypes.multipleInstancesOf(LCView.class))
                .join();

    }
}
```

One very simple way to replace these in-process calls will be to introduce some form of remote procedure call (RPC). Now our application looks like this:

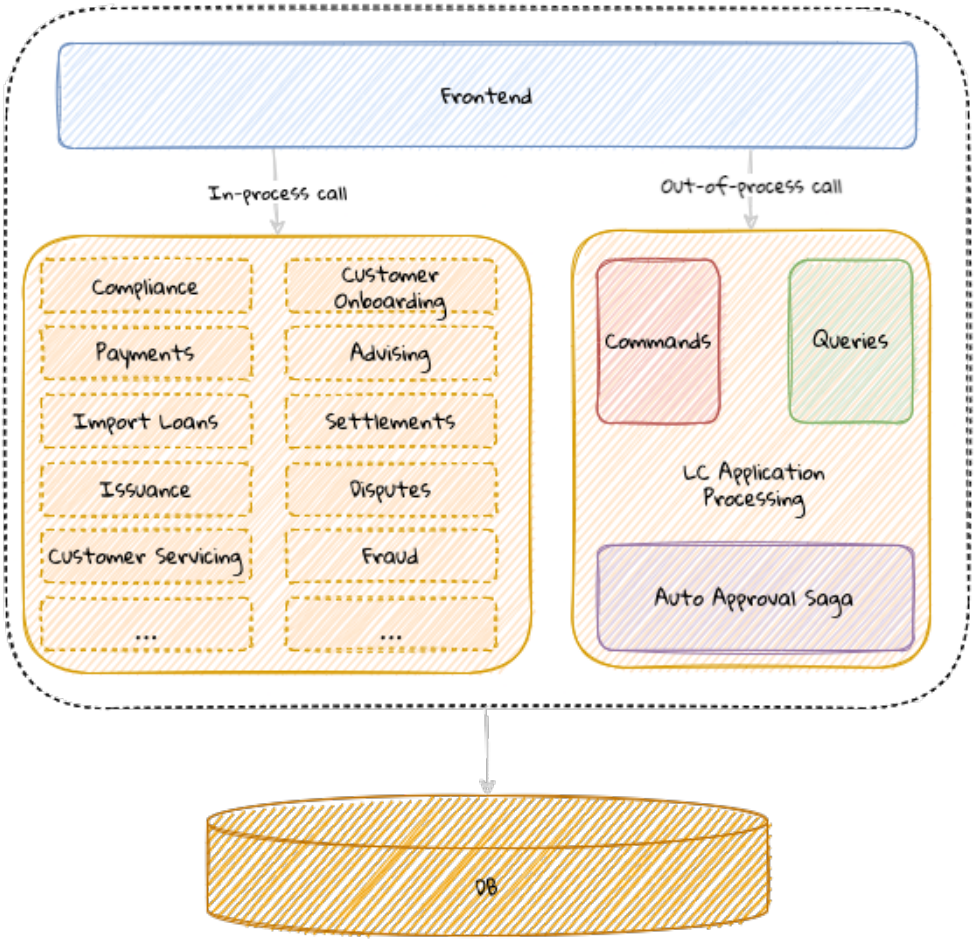


Figure 2. Remote interaction with the frontend introduced

When working with in-process interactions, we are simply invoking methods on objects within the confines of the same process. However, when we switch to using out-of-process calls, we have quite a few considerations. These days when working with remote APIs, we have several popular choices in the form of JSON-based web services, GraphQL, gRPC, etc. While it is possible to make use of a completely custom format to facilitate the communication, DDD advocates the use of the open host service pattern using a published language that we covered in chapter 9. Even with the open host service style of communication, there are a few considerations, some of which we discuss here:

Protocol options

There are several options available to us when exposing remote APIs. These days using a JSON-based API (often labeled as REST) seems to be quite popular. However, this isn't the only option available to us. In a resource-based approach, the first step is to identify a resource (noun) and then map the interactions (verbs) associated with the resource as a next step. In an action-based approach, the focus is on the actions to be performed. Arguably, REST takes a resource-based approach, whereas graphQL, gRPC, SOAP, etc. seem to be action-based. Let's take an example of an API where we want to submit an LC. In a RESTful world, this may look something like below:

```
# Start a new LC application
curl POST /lc-applications/start \
        -d '{"applicant-id": "8ed2d2fe", "clientReference": "Test LC"}' \
        -H 'content-type:application/vnd.lc-application.v2+json'
```

whereas with a graphQL implementation, this may look like:

```
mutation StartLCApplication {
  startLCApplication(applicantId: "8ed2d2fe",
                     clientReference: "Test LC") {
    lcApplicationId
  }
}
```

In our experience, designing APIs using REST does result in some form of dilution when attempting to mirror the language of the domain — because the focus is first and foremost on resources. Purists will be quick to point out that the example above is not RESTful because there is no resource named `start`. Our approach is to place more importance to remaining true to the ubiquitous language as opposed to being dogmatic about adherence to technical purity.

Transport format

Here we have two broad choices: text-based (for example, JSON or XML) versus binary (for example, protocol buffers or avro). If non-functional requirements (like performance) are met, our preference is to use text-based protocols as a starting point, because it can afford the flexibility of not needing any additional tools to visually interpret the data (when debugging).

When designing a remote API, we have the option of choosing a format that enforces a schema (for example, protocol buffers or avro) or something less formal like plain JSON. In such cases, in order to stay true to the ubiquitous language, the process may have to include additional governance in the form of more formal design and code reviews, documentation, etc.

Compatibility and versioning

As requirements evolve, there will be a need to enhance the interfaces to reflect these changes. This will mean that our ubiquitous language will also change over time, rendering old concepts to become obsolete. The general principle is to maintain backwards compatibility with consumers for as long as possible. But this does come with a cost of having to maintain old and new concepts together — leading to a situation where it can become hard to tell what is relevant versus what is not. Using an explicit versioning strategy can help in managing this complexity to an extent — where newer versions may be able to break backwards compatibility with older ones. But it is also not feasible to continue supporting a large number of incompatible versions indefinitely. Hence, it is important to make sure that the versioning strategy makes deprecation and retirement agreements explicit.

Error handling

REST APIs

Leonard Richardson coined the notion of a maturity model for HTTP-based REST APIs with each level being more mature than the preceding one:

1. **Resources**: TODO

2. **HTTP Verbs**: TODO
3. **Hypermedia controls**: TODO

Most web service based solutions that claim to be RESTful seem to stop at level 2. Roy Fielding, the inventor or REST seems to claim that REST APIs must be hypertext-driven[1]. In our opinion, the use of hypertext controls in APIs allows them to become self-documenting and thereby promotes the use of the ubiquitous language more explicitly. More importantly, it also indicates what operations are applicable for a given resource at that time in its lifecycle. For example, let's look at a sample response where all pending LC applications are listed:

```
GET /lc-applications?status=pending HTTP/1.1
Content-Type: application/json

HTTP/1.1 200 OK
Content-Type: application/prs.hal-forms+json
{
  "_embedded" : {
    "lc-applications" : [
      {
       "clientReference" : "Test LC",
       "_links" : {
         "self" : {
           "href" : "/lc-applications/582fe5f8"
         },
         "submit" : {
           "href" : "/lc-applications/582fe5f8/submit"
         }
       }
      },
      {
       "clientReference" : "Another LC",
       "_links" : {
         "self" : {
           "href" : "/lc-applications/7689da3e"
         },
         "approve" : {
           "href" : "/lc-applications/7689da3e/approve"
         },
         "reject" : {
           "href" : "/lc-applications/7689da3e/reject"
         }
       }
      }
    ]
  }
}
```

In the example above, the first LC application needs to be **submit**ted, whereas the second application needs to either be **approved** or **rejected** (presumably because it has already been

submitted). Notice how the response also does not need to include a `status` attribute so that they can use this to deduce which operations are relevant for LC application at that time. While this may be a subtle nuance, we felt that it is valuable to point out in the context of our DDD journey.

We have looked at a few considerations when moving from an in-process out-of-process API. There are quite a few other considerations, specifically pertaining to non-functional requirements (such as performance, resilience, error handling, etc.) We will look at these in more detail in Chapter 11.

Now that we have a handle on how we can work with APIs that interact with the front-end, let's look at how we can handle event publication and consumption **remotely**.

Changes for event interactions

Currently, our application publishes and consumes domain events over an in-process bus that the Axon framework makes available.

We publish events when processing commands:

Event publishing when processing a command

```
class LCApplication {

    // Boilerplate code omitted for brevity
    @CommandHandler
    public LCApplication(StartNewLCApplicationCommand command) {
        //...
        AggregateLifecycle.apply(new LCApplicationStartedEvent(command.getId(),
                command.getApplicantId(), command.getClientReference(), LCState.DRAFT
));
    }
}
```

and consume events to expose query APIs:

Event consumption to populate the query store

```
class LCApplicationSummaryEventHandler {

    // Boilerplate code omitted for brevity

    @EventHandler
    public void on(LCApplicationStartedEvent event) {
        //...
    }
}
```

In order to process events remotely, we need to introduce an explicit infrastructure component in the form of an event bus. Common options include message brokers like ActiveMQ, RabbitMQ or a distributed event streaming platform like Apache Kafka. Application components can continue to publish and consume events as before — only now they will happen using an out-of-process

invocation style. Logically, this causes our application to now look something like this:

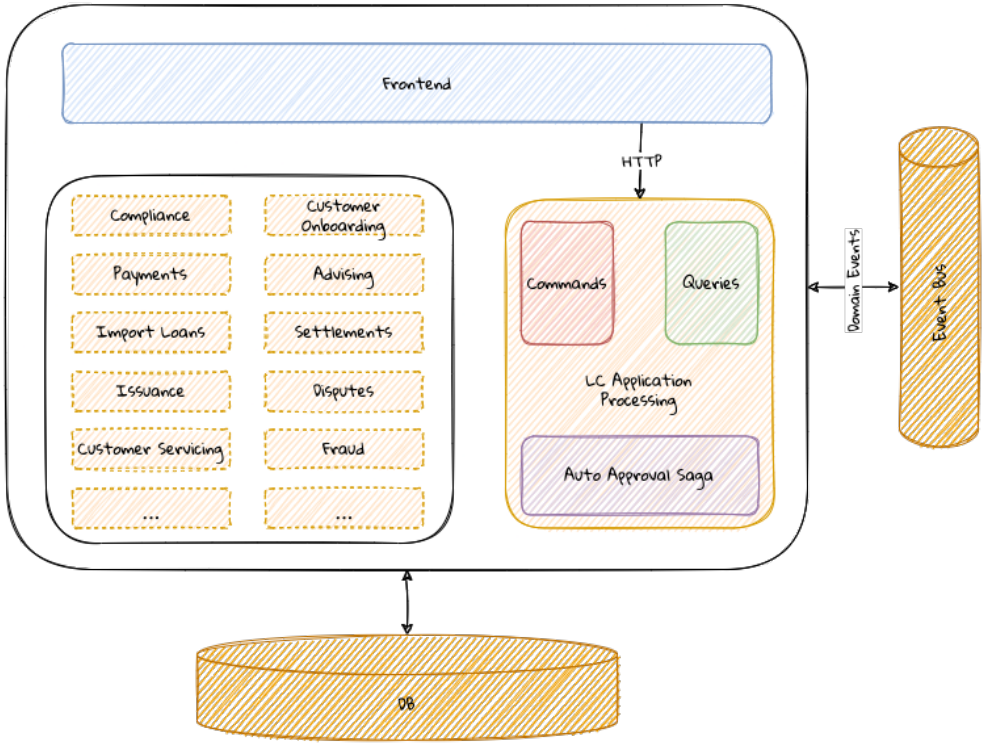


Figure 3. Out of process event bus introduced

When working with events within the confines of a single process, assuming synchronous processing (event publishing and consumption on the same thread), we do not encounter a majority of problems that only become apparent when the publisher and the consumer are distributed across multiple processes. Let's examine some of these in more detail next.

Atomicity guarantees

Previously, when the publisher processed a command by publishing an event and the consumer(s) handled it, transaction processing occurred as a single atomic unit as shown here:

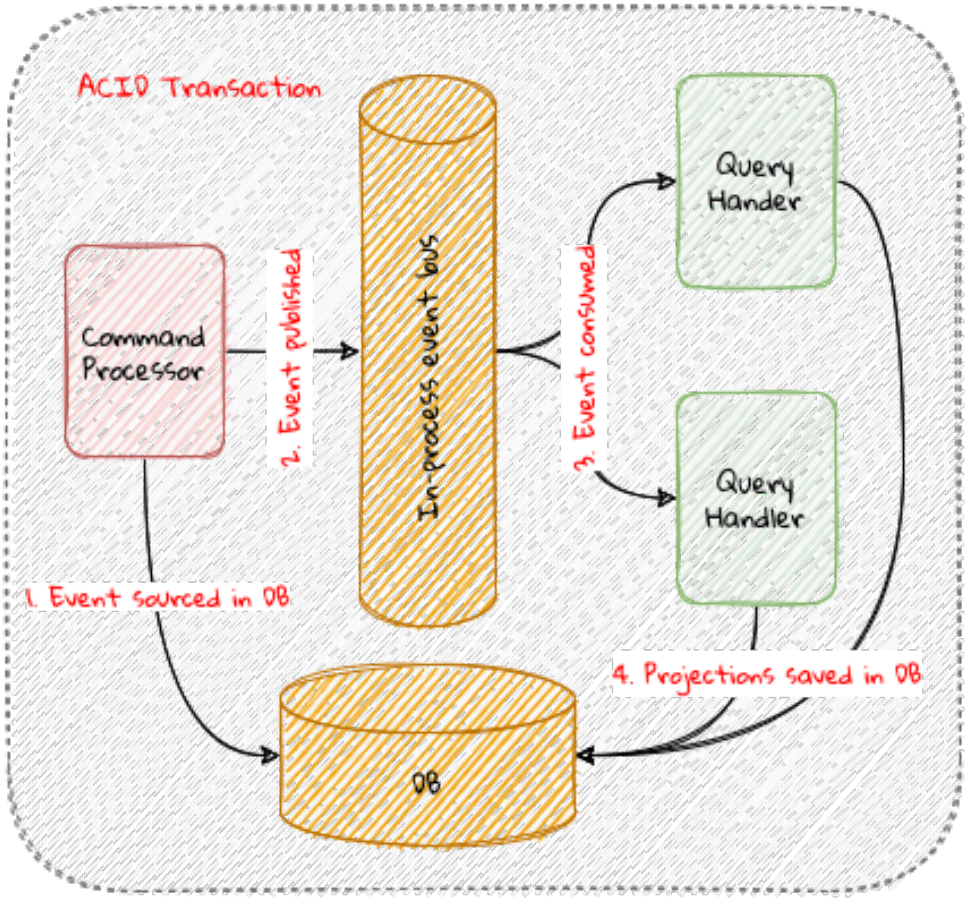


Figure 4. ACID transaction processing within the monolith

Notice how all the highlighted operations in the diagram above happen as part of a single database transaction. This allowed the system to be strongly consistent end-to-end. When the event bus is distributed to work within its own process, atomicity cannot be guaranteed like it was previously. Each of the above numbered operations work as independent transactions. This means that they can fail independently, which can lead to data inconsistencies.

To solve this problem, let's look at each step in the process in more detail, starting with command processing as shown here:

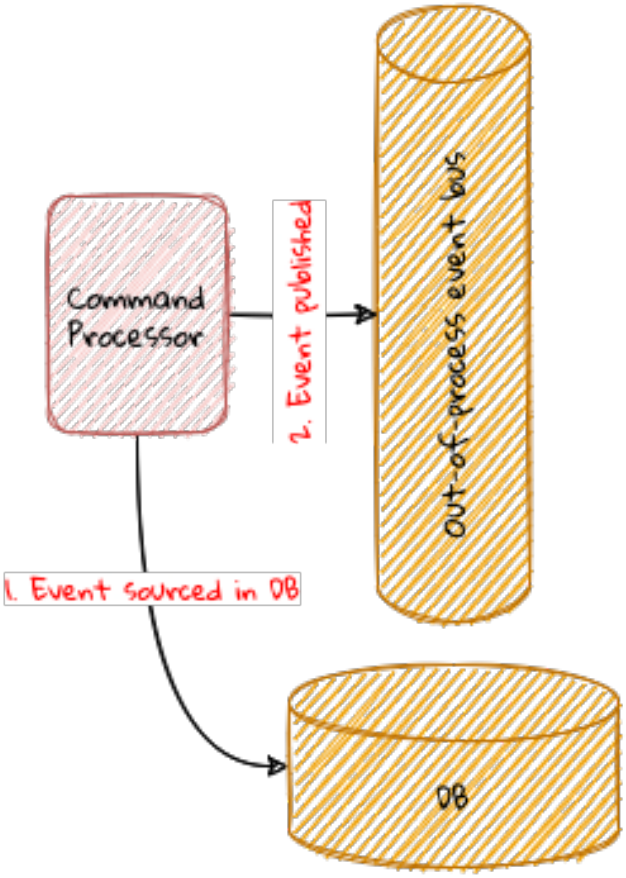


Figure 5. Command processing transaction semantics

Consider the situation where we save to the database, but fail to publish the event. Consumers will remain oblivious of the event occurring and become inconsistent. On the flip side, if we publish the event but fail to save in the database, the command processing side itself becomes inconsistent. Not to mention that the query side now thinks that a domain event occurred, when in fact it did not. Again, this leads to inconsistency. This **dual-write** problem is fairly common in distributed event-driven applications. If command processing has to work in a foolproof manner, saving to the database and the publishing to the event bus have to happen atomically - both operations should succeed or fail in unison. Here are a few solutions that we have used to solve this issue (in increasing order of complexity):

1. **Do nothing**: It is arguable that this approach is not really a solution, however it may be the only placeholder until a more robust solution is in place. While it may be puzzling to see this being listed as an option, we have seen several occasions where this is indeed how event-driven systems have been implemented. We leave this here as a word of caution so that teams become cognizant of the pitfalls.
2. **Transaction synchronization**: In this approach, multiple resource managers are synchronized in a way that failure in any one system, triggers a cleanup in the others where the transaction has already been committed. It is pertinent to note that this may not be foolproof in that it may lead to cascading failures.

The spring framework provides support for this style of behavior through the `TransactionSynchronization` interface. Please refer to the framework documentation for more details. Needless to say, this interface should not be used without careful consideration to business requirements.

3. **Distributed transactions**: Another approach is to make use of distributed transactions and two-phase commit[2]. Typically, this functionality is implemented using pessimistic locking on the underlying resource managers (databases) and may present scaling challenges in highly concurrent environments.
4. **Transactional outbox**: All the above methods are not completely foolproof in the sense that there still exists a window of opportunity where the database and the event bus can become inconsistent (this is true even with two-phase commits). One way to circumvent this problem is by completely eliminating the dual-write problem. In this solution, the command processor writes to its database and the intended event to an *outbox* table in a local transaction. A separate poller component polls the outbox table and writes to the event bus. Polling can be computationally intensive and may again lead back to the dual write problem because the poller has to keep track of the last written event. This may be avoided by making event processing idempotent on the consumer so that processing duplicate events do not cause issues. In extremely high Another way to mitigate this issue is to use a change data capture tool (like Debezium[3]). Most modern databases ship with tools to make this easier and may be worth exploring. Here is one way to implement this using the *transactional outbox pattern* as shown here:

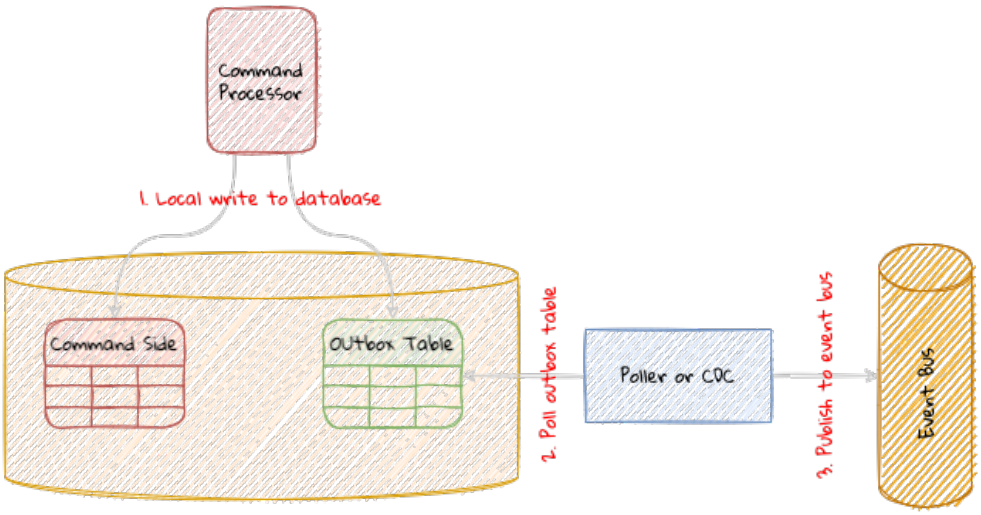


Figure 6. Transactional outbox

The transactional outbox is a robust approach to dealing with the dual write problem. But it also introduces a non-trivial amount of operational complexity. In one of our previous implementations, we made use of the transactional synchronization to ensure that we never missed writes to the database. We also ensured that the event bus was highly available through redundancy on both the compute and storage tiers, and most importantly by avoiding **any** business logic on the event bus.

Delivery guarantees

When it comes to message delivery semantics, there are three basic categories:

1. **At-most once** delivery means that each message may be delivered once or not at all. This style of delivery is arguably the easiest to implement because the producer creates messages in a fire and forget fashion. This may be okay in environments where loss of some messages may be tolerated. For example, data from click-stream analytics or logging might fall in this category.
2. **At-least once** delivery means that each message may be delivered more than once with no messages being lost. Undelivered messages are retried to be delivered — potentially infinitely. This style of delivery may be required when it is not feasible to lose messages, but where it may be tolerable to process the same message more than once. For example, analytical environments may tolerate duplicate message delivery or have duplicate detection logic to discard already processed messages.
3. **Exactly once** delivery means that each message is delivered exactly once without either being lost or duplicated. This style of message delivery is extremely hard to implement and a lot of solutions may approach exactly once semantics with some implementation help from the consumers where duplicate messages are detected and discarded with the producer sticking to at least once delivery semantics.

For the purposes of domain event processing, we will obviously prefer to have exactly once processing semantics. However, given the practical difficulties guaranteeing *exactly once* semantics, it is not unusual to approach exactly once processing by having the consumers process events in an idempotent manner or designing events to make it easier to detect errors. For example, consider a `MonetaryAmountWithdrawn` event, which includes the `accountId` and the `withdrawalAmount`. This event may carry an additional `currentBalance` attribute so that the consumer may know if they are out of sync with the producer when processing the withdrawal. Another way to do this might be for the consumer to keep track of the last "n" events processed. When processing an event, the consumer can check if this event has already been processed. If so, they can detect it as a duplicate and simply discard it. All the above methods again add a level of complexity to the overall system. Despite all these safeguards, consumers may still find themselves out of sync with the system of record (the command side that produces the event). If so, as a last resort, it may be required to use a partial or full event replays which was discussed in Chapter 7.

Ordering guarantees

1. No order
2. Order per aggregate
3. Global ordering

Durability guarantees

Failure processing

Scaling the event bus

Having done this allows us to actually extract the LC application processing component into its own independently deployable unit, which will look something like this:

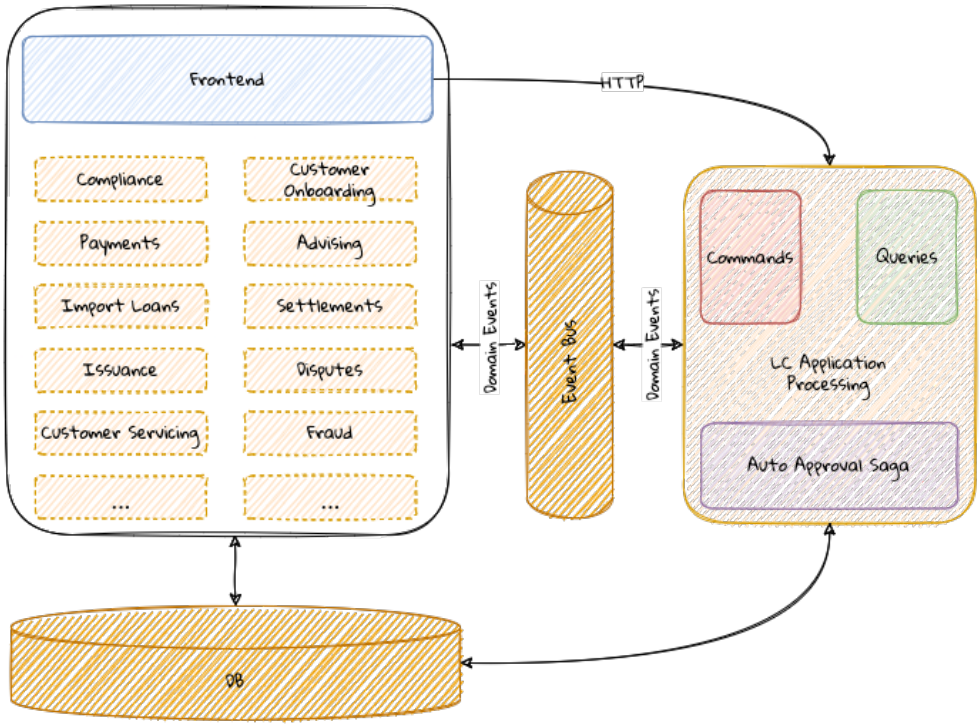


Figure 7. LC Application processing deployed independently

Changes for database interactions

While we have extracted our component into its own unit, we continue to be coupled at the database tier.

Historic data migration

Cutover

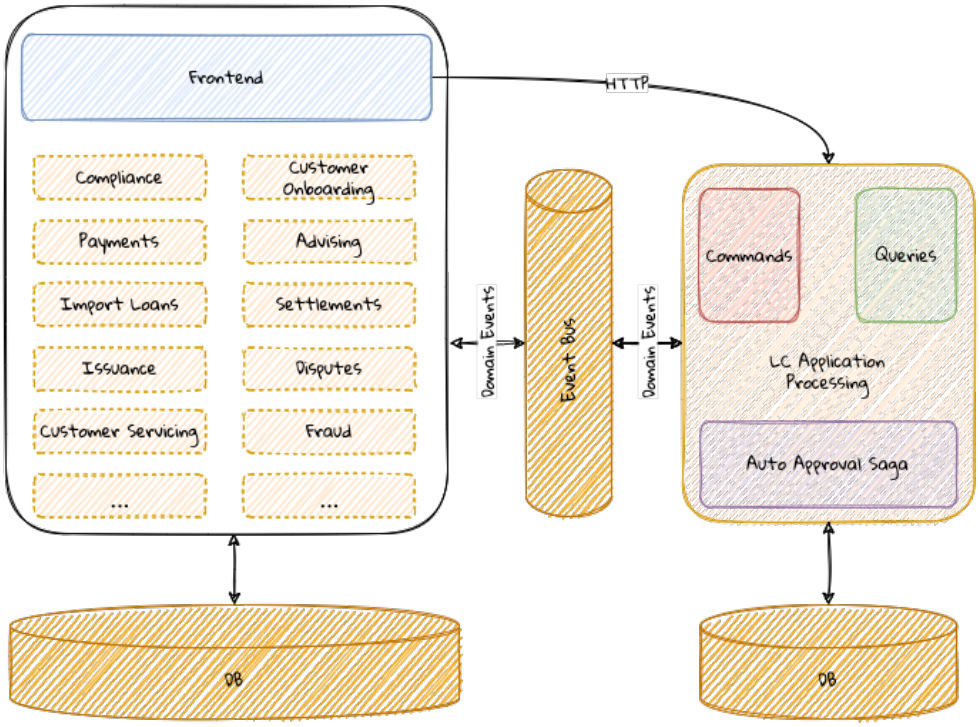
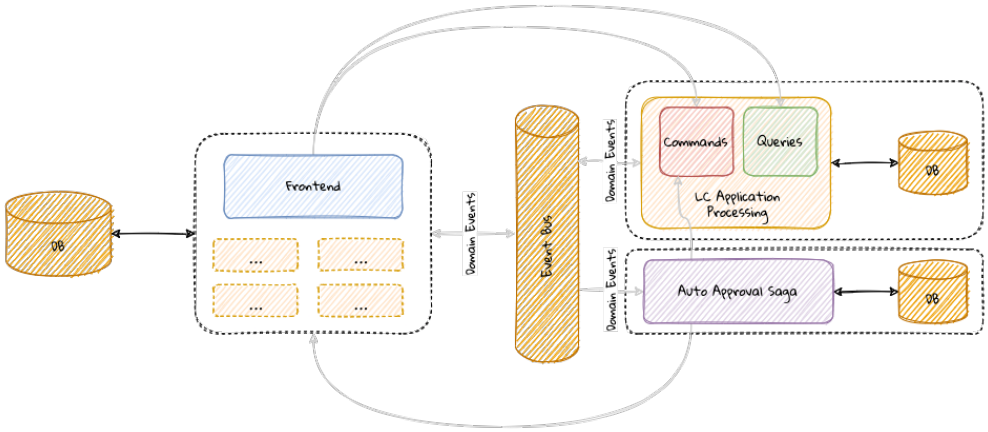


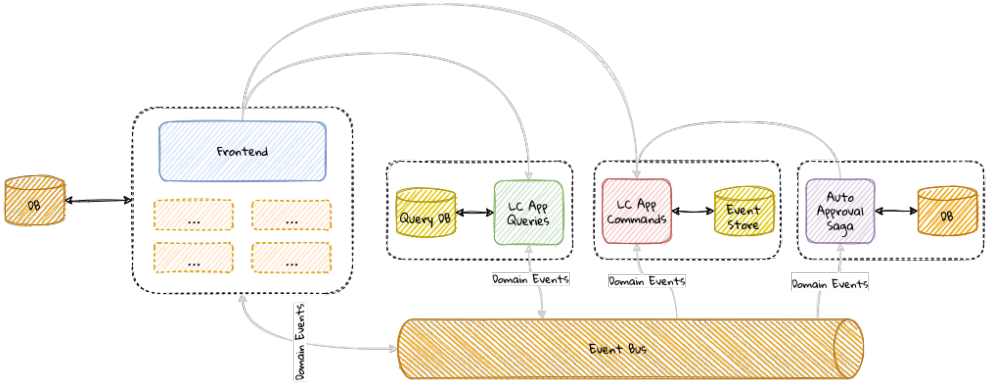
Figure 8. Independent data persistence

Potential next steps

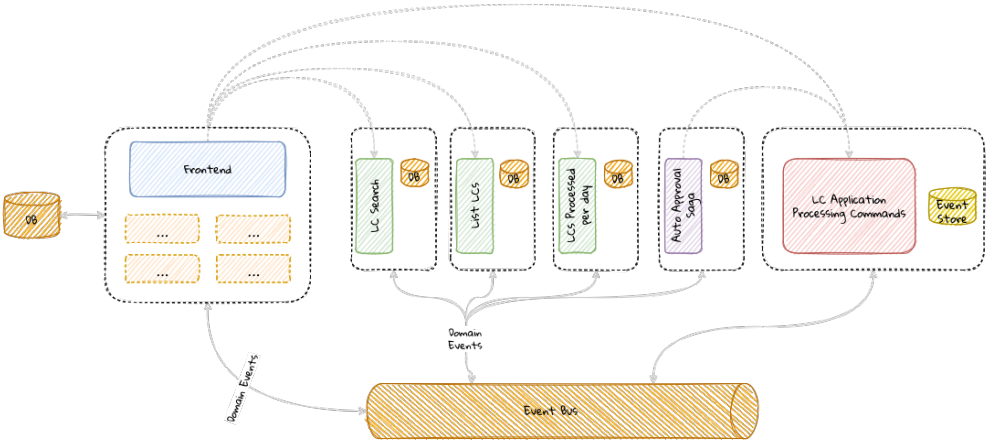
Sagas as standalone components



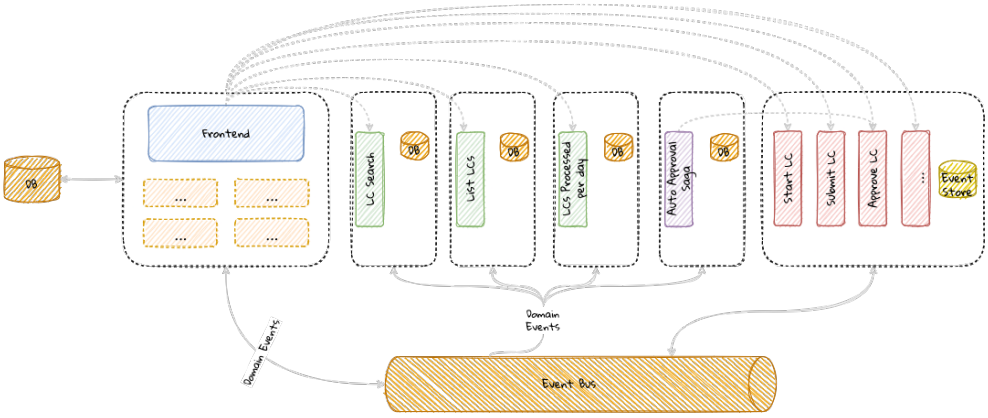
Commands and queries as standalone components



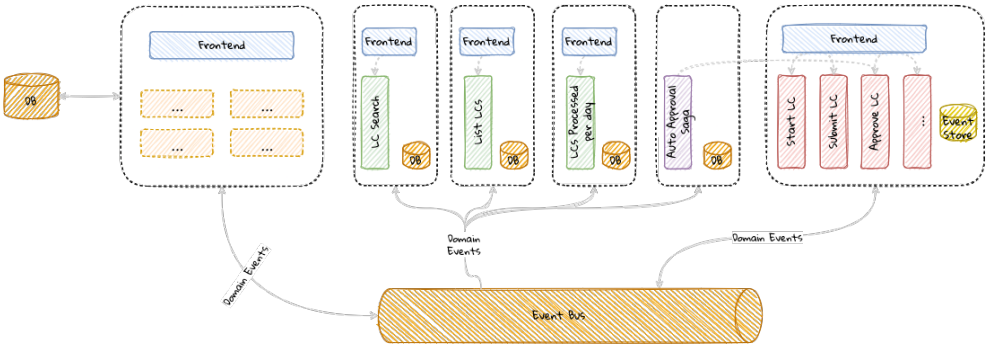
Distributing individual query components



Even more fine-grained distribution



Customer experiences and frontends



Non-technical implications of distribution

Team topologies

Tech stack

Technical implications of distribution

Performance

Resilience

Exception handling

> Everything fails all the time.
>
> — Werner Vogels, CTO -- Amazon Web Services

Unexpected failures in software systems are bound to happen. Instead of expending all our energies trying to prevent them from occurring, it is prudent to embrace and design for failure as well. Let's look at the scenario in the `AutoApprovalSaga` and identify things that can fail:

```
class AutoApprovalSaga {
    //...
    @SagaEventHandler(associationProperty = "lcApplicationId")
    public void on(ProductLegalityValidatedEvent event) {
        //..
        productLegalityValidated = true;
        if (productValueValidated && applicantValidated) {
               gateway.send(new ApproveLCApplicationCommand(lcApplicationId)); ①
        }
    }
}
```

① When dispatching commands, we have a few styles of interaction with the target system (in this case, the LC application bounded context):

- **Fire and forget**: This is the style we have used currently. This style works best when system scalability is a prime concern. On the flip side, this approach may not be the most reliable because we do not have definitive knowledge of the outcome.
- **Wait infinitely**: We wait infinitely for the dispatch and handling of the `ApproveLCApplicationCommand`.
- **Wait with timeout**: We wait for a certain amount of time before concluding that the handling of the command has likely failed.

Which interaction style should we use? While this decision appears to be a simple one, it has quite a few, far-reaching consequences:

- If command dispatching itself fails, the `CommandGateway#send` method will fail with an exception. Given that the CommandGateway is an infrastructure component, this will happen because of technical reasons (like network blips, etc.)
- If the command handling for the `ApproveLCApplicationCommand` fails, and we will not know about it because the `#send` method does not wait for handling to complete. One way to mitigate that problem is to wait for the command to be handled using the `CommandGateway#sendAndWait` method. However, this variation waits infinitely for the handler to complete — which can be a scalability concern.
- We can choose to only wait

Recovery

Automated recovery

Manual recovery

Compensating actions

Retries

Integration strategies

Not yet finalized

Transitional architecture

Understanding the costs of distribution

Handling exceptions

Testing the Overall System

Compatibility

[1] https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[2] https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html

[3] https://debezium.io/