

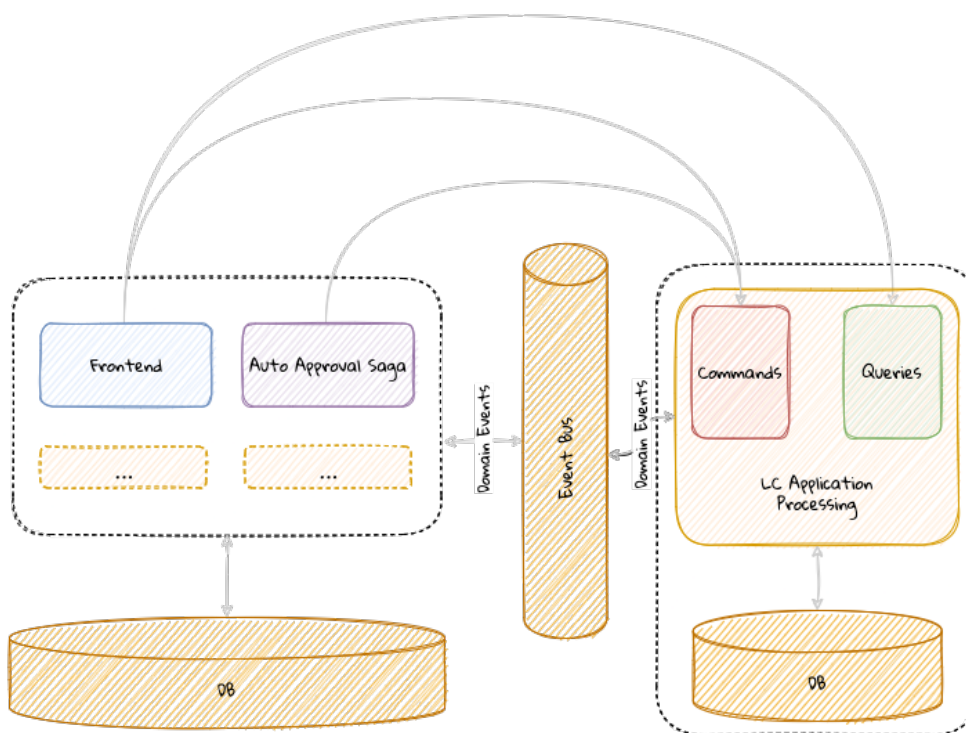
# Table of Contents

Distributing into finer grained components .....	1
Potential next steps .....	1
Saga as standalone components .....	1
Commands and queries as standalone components .....	2
Distributing individual query components .....	3
Even more fine-grained distribution .....	3
Effects on the domain model .....	4
Customer experiences and frontends .....	5
Where to draw the line? .....	5

## Distributing into finer grained components

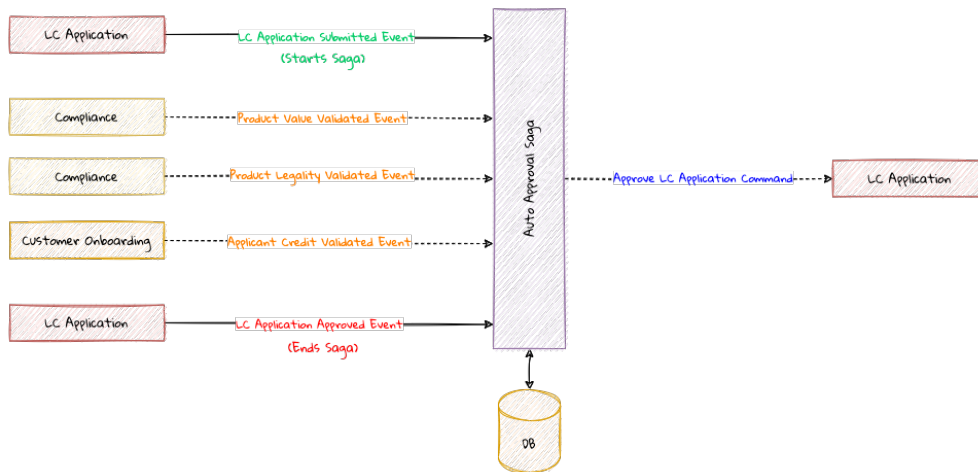
In this chapter, we will migrate the components developed thus far to adopt a serverless style of architecture.

### Potential next steps

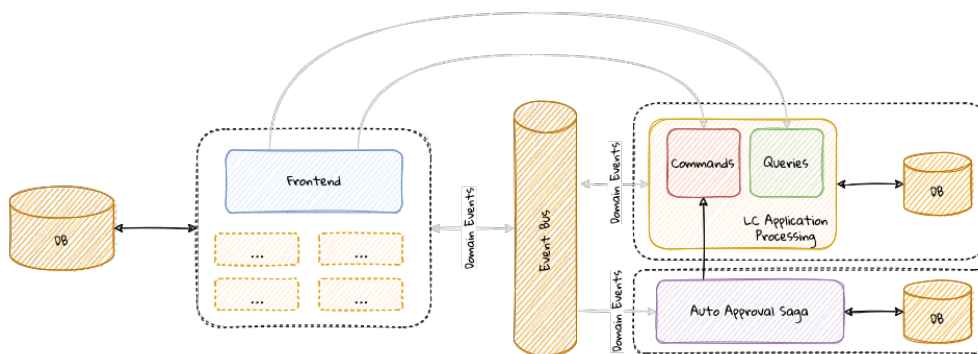


### Saga as standalone components

Currently, the `AutoApprovalSaga` component (discussed in detail in Chapter 8) works by listening to domain events as shown here:

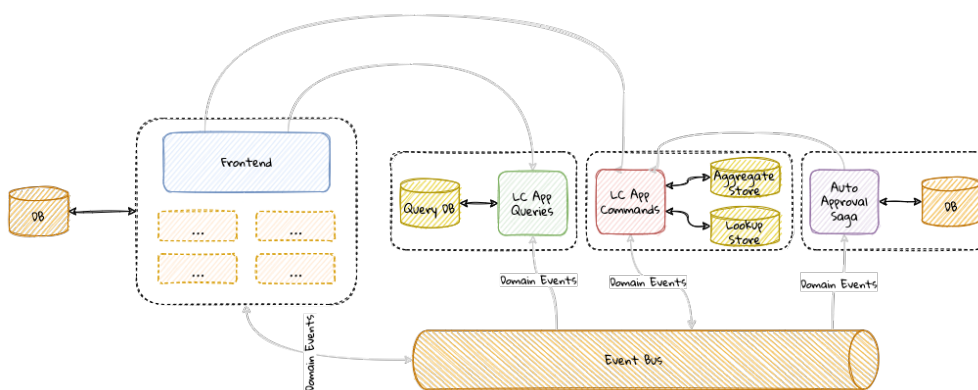


Given that these events are published by different bounded contexts onto the event bus, there is no need for `AutoApprovalSaga` to be embedded within the monolith. This means that it can be safely pulled out into its own deployable unit along with its private datastore. This means that our system now looks like the visual depicted here:



## Commands and queries as standalone components

As we have seen in the section on the [CQRS pattern](#) in earlier chapters, the primary benefit that we derive is gaining the ability to evolve and scale these components independently of each other. This is important because commands and queries have completely different usage patterns and thus require the use of distinct domain models. This makes it fairly natural to further split our bounded contexts along these boundaries. Thus far, the segregation is logical. A physical separation will enable us to truly scale these components independently as shown here:



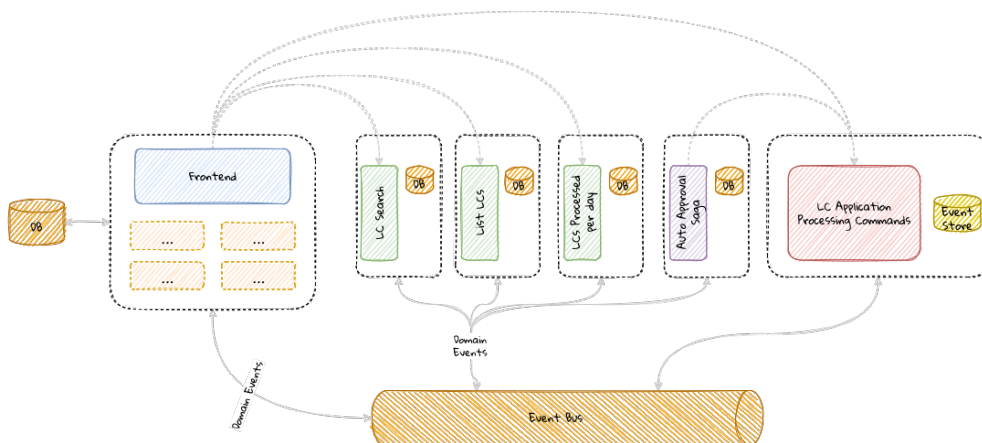
It is pertinent to note that the command processing component is now shown to have access to two distinct datastores:

- The **aggregate store**: which stores either an event-sourced or state-stored representation of aggregate state
- The **lookup store**: which can be used to store lookup data when performing business validations when processing commands. This is applicable when requiring to access data that is/cannot be stored as part of aggregate state.

The reason we bring this up is that we may have to continue making lookups for data that still remain in the monolith. To achieve full independence, this lookup data must also be migrated using techniques like a [historic event replay](#) (discussed in Chapter 7) or other conventional [data migration](#) techniques (discussed in Chapter 10).

## Distributing individual query components

At this point, we have achieved segregation along command and query boundaries. But we do not need to stop here. Each of the queries we service need not necessarily remain a single component. Let's consider an example where we need to implement a fuzzy LC search feature for the UI and a view of LC facts for analytical use cases. It is conceivable that these requirements may be implemented by a different set of teams, thereby necessitating the need for distinct components. Even if these are not distinct teams, the disparity in usage patterns may warrant the use of different persistence stores and APIs, again requiring us to look at implementing at least a subset of these as distinct components as shown here:



Owning domains should strive to create query APIs that exhibit the characteristics of a good [domain data product](#)<sup>[1]</sup>.

## Even more fine-grained distribution

At this stage, is there any further decomposition that is required and feasible? These days, whether rightfully or otherwise, the serverless architecture (specifically, *Functions-As-A-Service*) is arguably becoming quite the rage. As we pointed in Chapter 2, this means that we may be able to decompose our command side in a manner that each command becomes its own independently deployable unit (hence a bounded context). In other words, the `LCApplicationSubmitCommand` and the `LCApplicationCancelCommand` can be deployed independently.

But just because this is technically possible, should we do it? While it is easy to dismiss this as a passing fad, there may be good reasons to split applications along command boundaries:

- **Risk profile:** Certain pieces of functionality present a higher risk when changes are made. For example, submit an LC application may be deemed a lot more critical than the ability to cancel it. But that is not to say that *cancel* is unimportant. Being decoupled from *submit* allows *cancel* changes to be made with a lot less scrutiny. This may make it easier to innovate at pace for more experimental features with minimal fear of causing large disruptions.
- **Scalability needs:** Scaling needs can differ wildly for various commands in the system. For example, *submit* may need to scale a lot more than *cancel*. But being coupled will force us to treat them as equals, which can be inefficient.
- **Cost attribution:** Having fine-grained components allows us to more accurately measure the amount of effort and the resulting ROI dedicated to each individual command. This can make it easier to focus our efforts on the most critical functionality (the "core" of the core) and minimize waste.

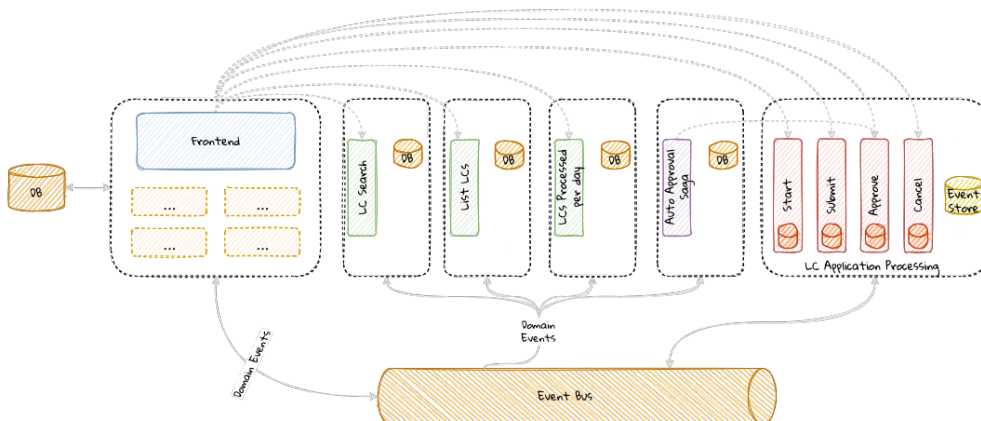
## Effects on the domain model

These finer grainer components are leading us to a point where it may appear that the deployment model is starting to have a big influence on the design. The fact that it is now feasible to deploy individual "tasks" independently, requires us to re-examine how we arrive at bounded contexts. For example, we started by working on the *LC Application Processing* bounded context. And our aggregate design was based on all functionality included in the scope of application processing. Now, our aggregate design can be a lot more fine-grained. This means that we can have an aggregate specifically for *submit* functionality and another for *cancel* as shown here:

[fine grained aggregate] | *fine-grained-aggregate.png*

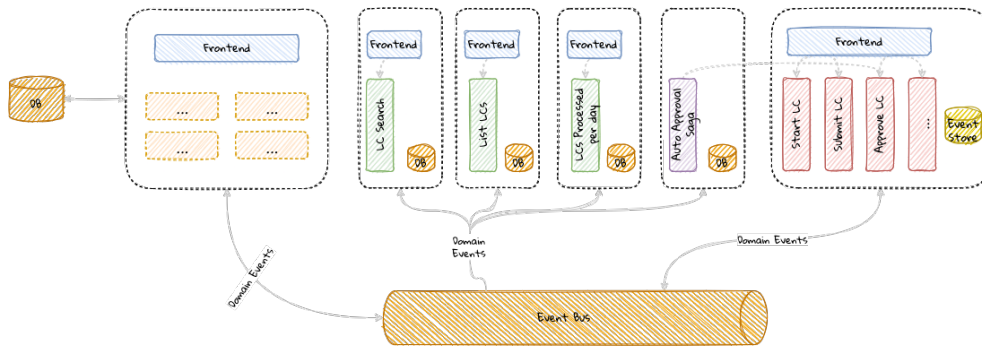
Figure 1. Fine-grained bounded contexts example

The most fine-grained decomposition may lead us to a bounded context per command. But that does not necessarily mean that we have to decompose the system this way. In the above example, we have chosen to create a single bounded context for the *submit* and *approve* commands. However, *start* and *cancel* have their own bounded contexts. The actual decision that you make in your own ecosystems will depend on maintaining a balance among reuse, coupling and other considerations that we discussed earlier. It is important to note that the aggregate labeled as **LCApplication**, although named identically, is distinct from a domain model perspective in its respective bounded context. The only attribute they will need to share is a **common identifier**. This leads our overall solution to look like the visual shown here:



It is pertinent to note that the commands functions continue to share a single event store, although they may make use of their own individual lookup stores.

## Customer experiences and frontends



## Where to draw the line?

[1] <https://martinfowler.com/articles/data-monolith-to-mesh.html#DomainDataAsAProduct>