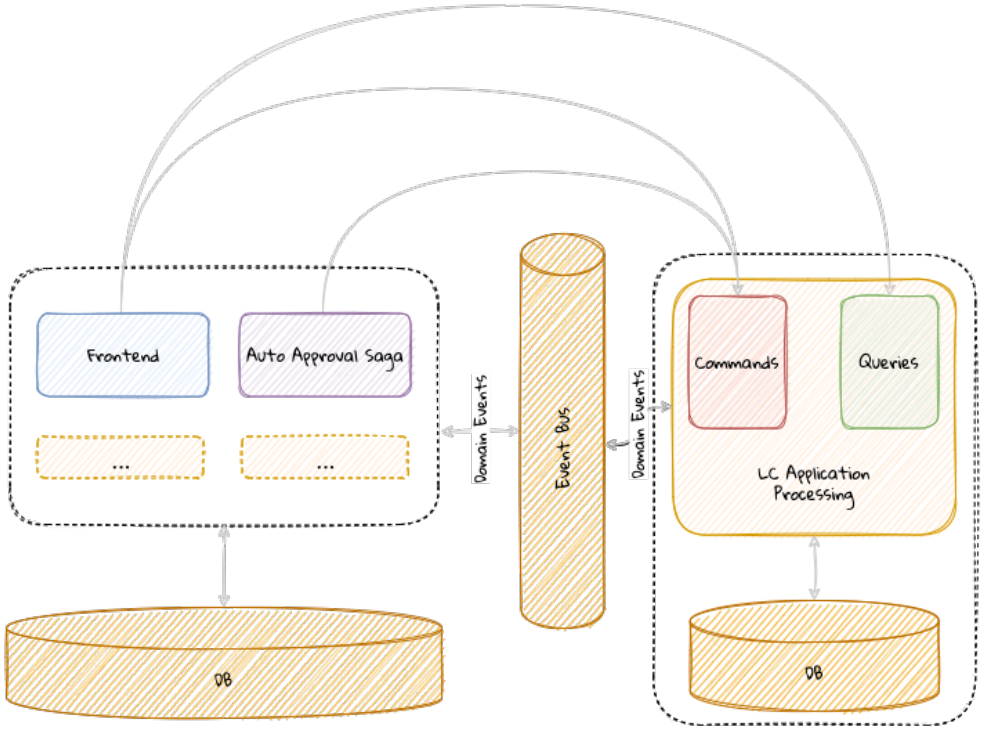
Table of Contents

Distributing into finer grained components

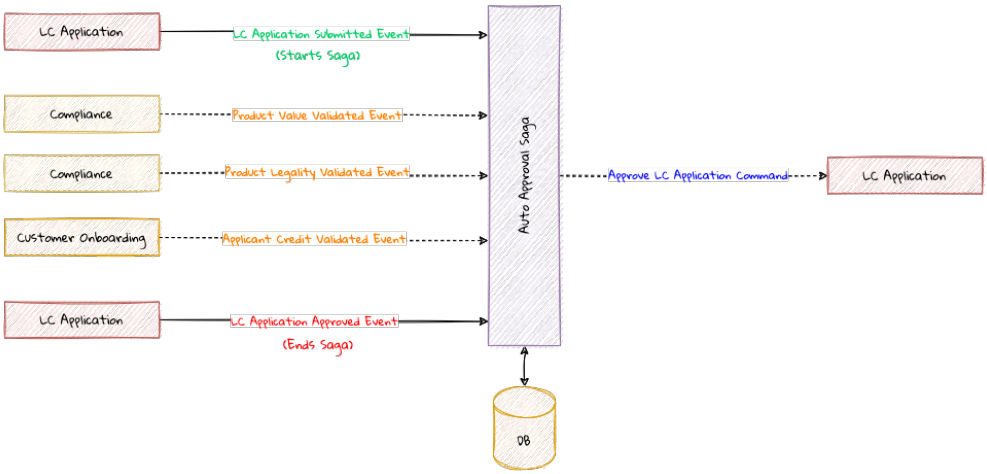
In this chapter, we will migrate the components developed thus far to adopt a serverless style of architecture.

Potential next steps

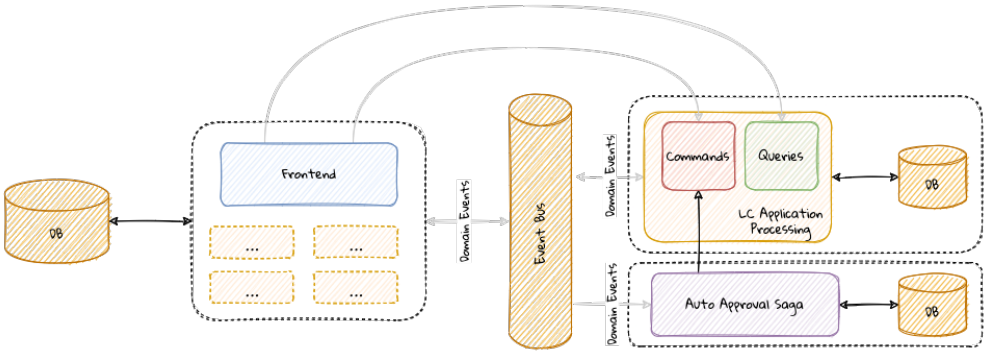


Saga as standalone components

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

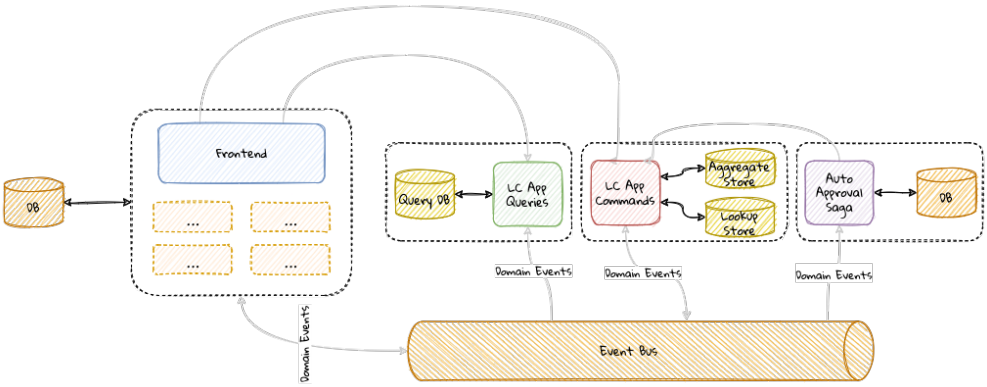


Given that these events are published by different bounded contexts onto the event bus, there is no need for `AutoApprovalSaga` to be embedded within the monolith. This means that it can be safely pulled out into its own deployable unit along with its private datastore. This means that our system now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



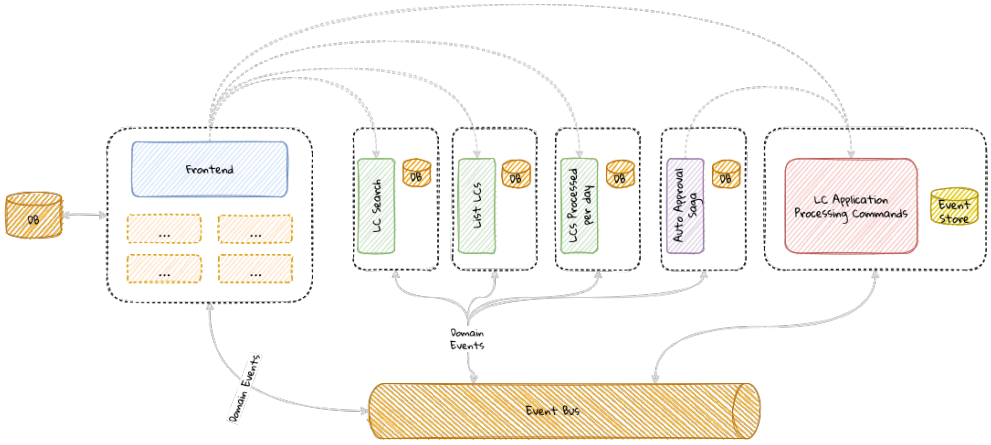
It is pertinent to note that the command processing component is now shown to have access to two distinct datastores:

- 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[1].

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 *start* functionality and another for *cancel* as shown here:

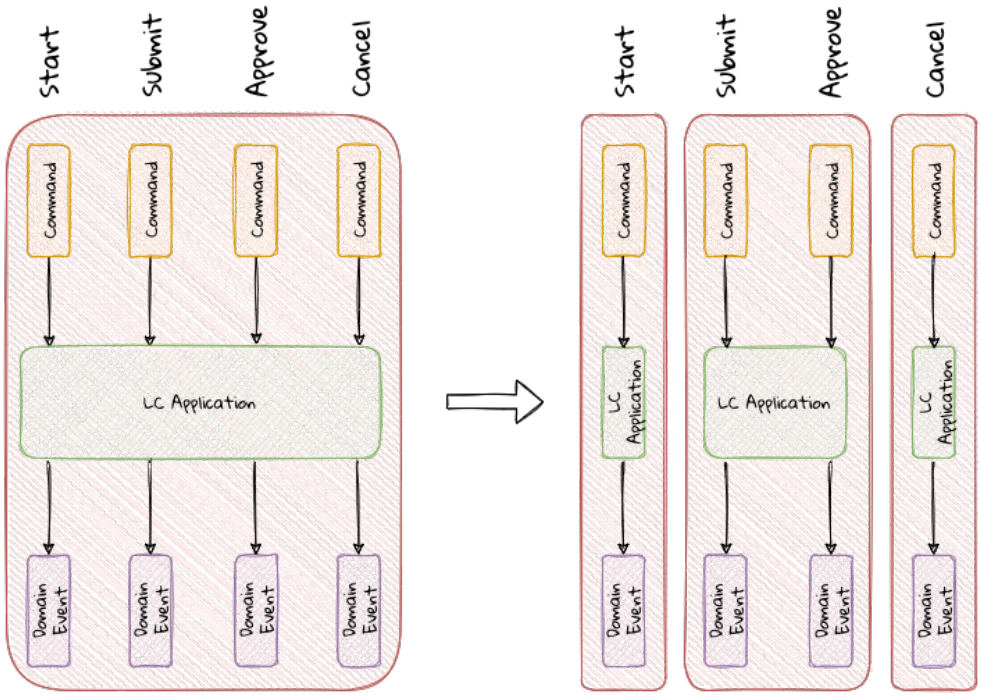


Figure 1. Fine-grained bounded contexts example

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, transactional

consistency 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**. If we choose to decompose the system into a bounded context per command, our overall solution will look like the visual shown here:

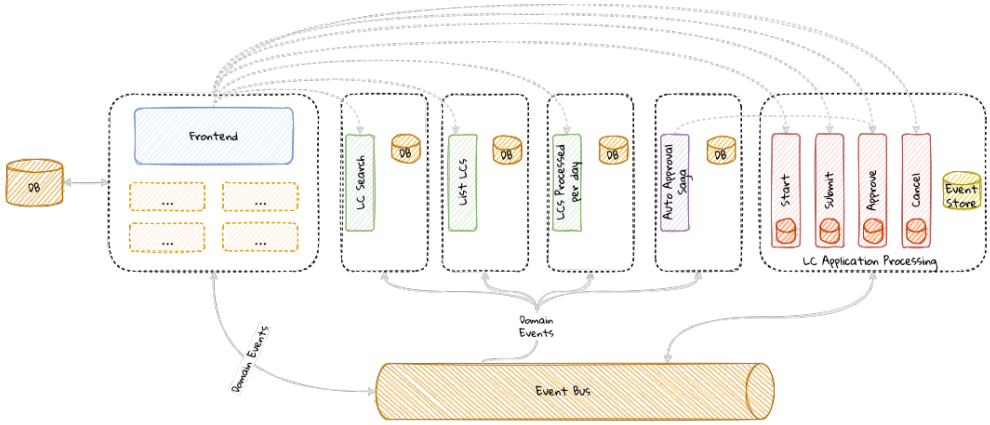


Figure 2. Decomposition per command

It is pertinent to note that the *command* functions continue to share a single event store, although they may make use of their own individual lookup stores. We understand that this decomposition likely feels unnecessary and forced. However, this does allow us to focus our energies on the ***core of the core***. For example, LC application processing may be our business differentiator. But an even more careful examination may reveal that it is our ability to *decision* LCs near real-time that is our real business differentiator. This means that it may be prudent to isolate that functionality from the rest of the system. In fact, doing so may enable us to optimize our business process without adding additional risk to the overall solution. While it is not strictly necessary to decompose the system in this way to arrive at such insights, the fine-grained decomposition may enable us to refine the idea of what is most important to our business. Having to share a persistent store can be a wrinkle to achieve complete independence. So a final decomposition may look something like what we show here:

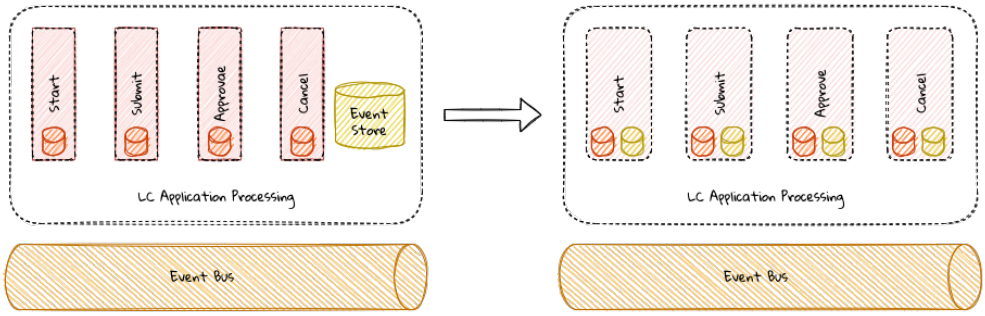


Figure 3. Command components with individual event stores.

Obviously, there is no free lunch! This fine-grained decomposition may require additional coordination and duplication of data among these components — to a point where it may not be attractive anymore. But we felt that it is important to illustrate the art of the possible.

Customer experiences and frontends

Thus far, we have focussed on decomposing and distributing the backend components while

keeping the frontend untouched as part of the existing monolithic system. It is worth considering breaking down the frontend to align more closely along functional boundaries. Patterns like micro-frontends[2][3] extend the concepts of microservices to the frontend. Micro-frontends promote team structures to support end-to-end ownership of a set of features. It is conceivable that a cross-functional, polyglot team owns both the experience (frontend) and the business logic (backend) functions eliminating communication overheads drastically (along the lines of the vertical slice architecture conversation in Chapter 2). Even if such a team organization where the frontend and backend being one team is not feasible in your current ecosystem, this approach still has many merits such as:

Increased Customer focus?

Autonomy/Technology agnostic

Decoupled code-bases allow for parallelization

Independent Deployment?

While there are many advantages in considering the micro-frontend approach, it does bring some challenges :

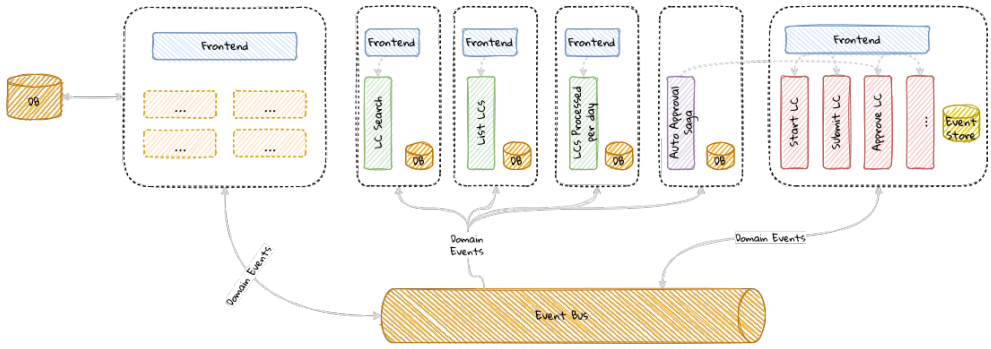
Deployment complexity - TODO: Mobile Apps? Single SPA

Build time integration

Duplication

Runtime Inefficiency (more code downloaded to browser)

Consistency (UI & User Experience, different framework versions)



Where to draw the line?

In general, the smaller the size of our bounded contexts, the easier it becomes to manage complexity. Does that mean we should decompose our system into as fine-grained a granularity as possible? On the other hand, having extremely fine-grained components can increase coupling among them to an extent where it becomes very hard to manage operational complexity. Hence, decomposing a system into collaborating components can be a bit tricky, seeming to work more like an art, rather than an exact science. There is no right or wrong answer here. In general, if things feel and become painful, you most likely got it more wrong than right. Here are some non-technical heuristics that might help guide this process:

- **Existing organization boundaries**: Look to align along current org structures. Identify which applications your business unit/department/team already owns and assign responsibilities in a manner that causes minimal disruption.
- **Domain expert roles and responsibilities**: What work do your domain expert carry out? What enables them to do their work with the least friction possible?
- **Change in vernacular**: Look for subtle changes in the usage of common terms. Does someone call something that is/feels the same in the physical world by different names? For example, a credit card may be called "plastic", "payment instrument", "account" by different people or the same people in a different context.
- **Existing (modular/monolithic/distributed) applications**: How are your current applications segregated logically? How are they segregated physically? This might provide some inspiration.

But what if one or more of the above are wrong/cumbersome/suboptimal? In such a case, our work as developers/architects is a bit more involved. It is not uncommon to get domain boundaries wrong. Come up with an initial breakdown that seems to make more sense and apply a series of *what if* questions to assess suitability. If the reasoning is able to stand up to scrutiny by domain experts, architects and other stakeholders, you might have your answer. If you do choose to go this route, it may be prudent to adjust existing organization structures to match your proposed architecture. This will help reduce friction (in other words, you have applied the **inverse Conway maneuver**[4]).

Despite all our due diligence and noble intentions, it is still possible to get these boundaries wrong. Or a change in business priorities or competitor offerings may render decisions that appeared perfectly valid at the time to become incorrect. Instead of looking to arrive at the perfect decomposition, it might be prudent to embrace change and invest in building designs that are flexible while being prepared to evolve and refactor the architecture iteratively. The book on building evolutionary architectures[5] has some great advice on how to do precisely that.

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

[2] https://micro-frontends.org/

[3] https://martinfowler.com/articles/micro-frontends.html

[4] https://www.thoughtworks.com/en-us/radar/techniques/inverse-conway-maneuver

[5] https://evolutionaryarchitecture.com/