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:

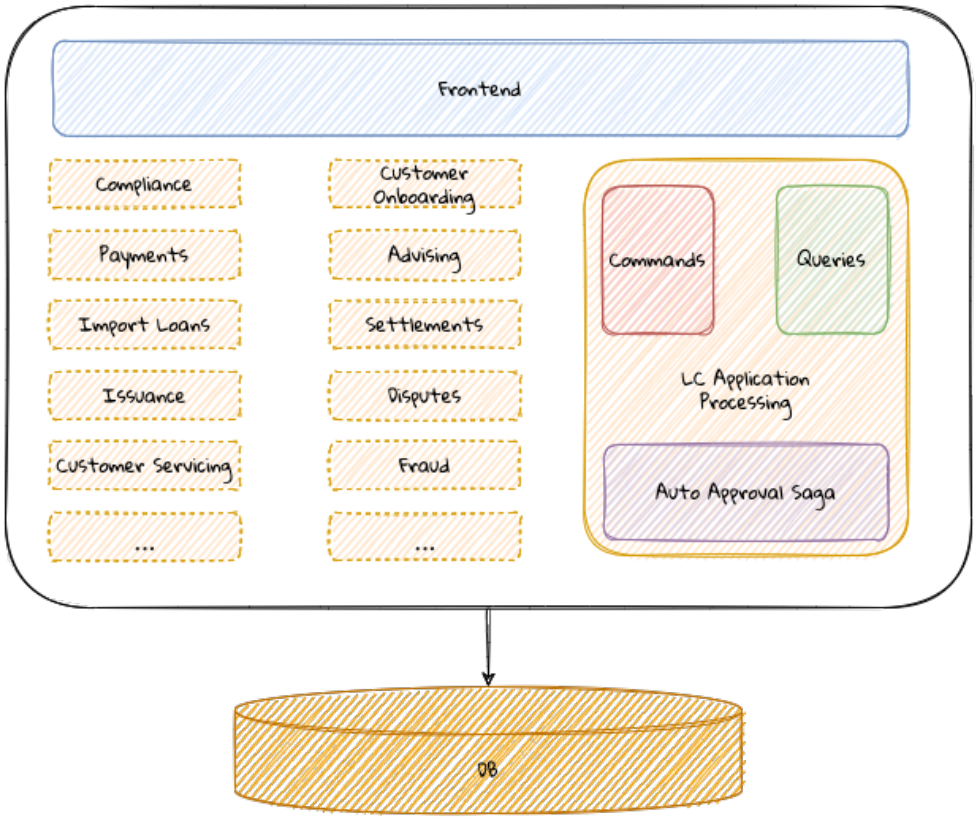


Figure 1. Current view of the LC application monolith

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:

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). For example, JSON-based web services, GraphQL, gRPC, etc. In each case, we will be making use of the open host service pattern using a published language that we covered in Chapter 9.

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

Now our application looks like this:

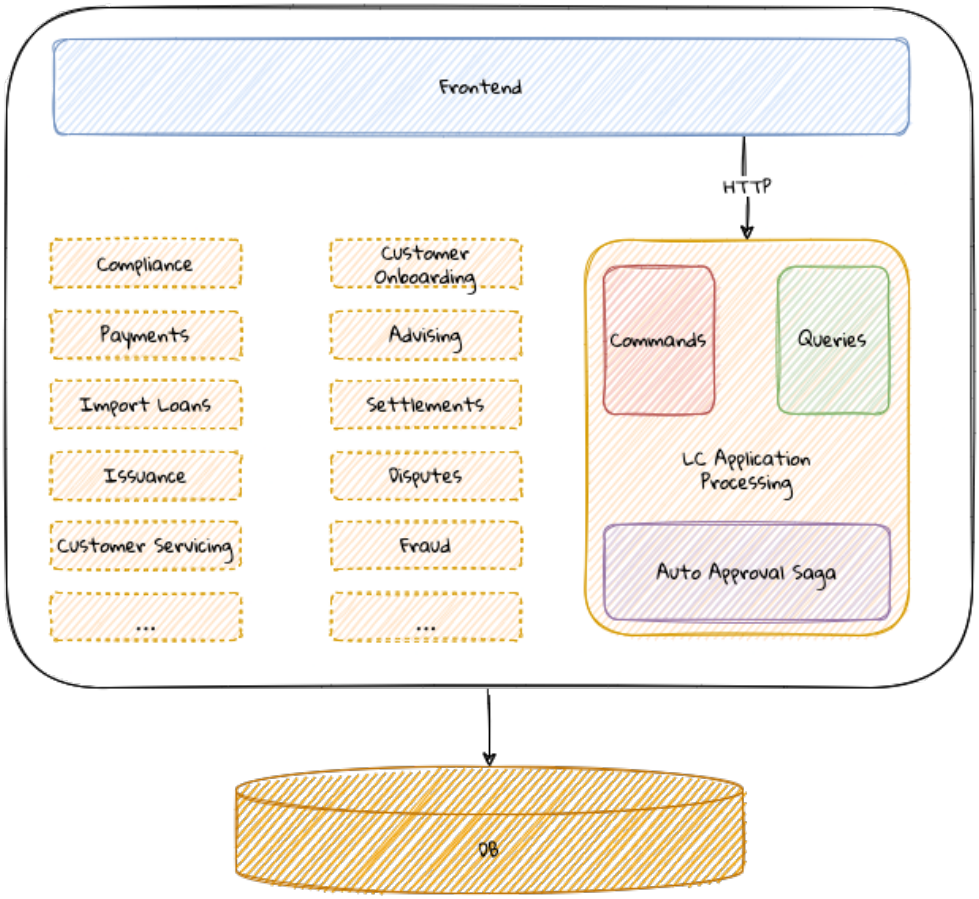


Figure 2. Remote interaction with the frontend introduced

Remoting options

TODO: when to use REST, GraphQl or gRPC

Next, we need to look at how we can handle event publication and consumption remotely.

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:

```
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 when constructing query stores:

```
class LCApplicationSummaryEventHandler {

    // Boilerplate code omitted for brevity

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

This can be done by introducing an explicit infrastructure component in the form of a remote event bus. 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 like this:

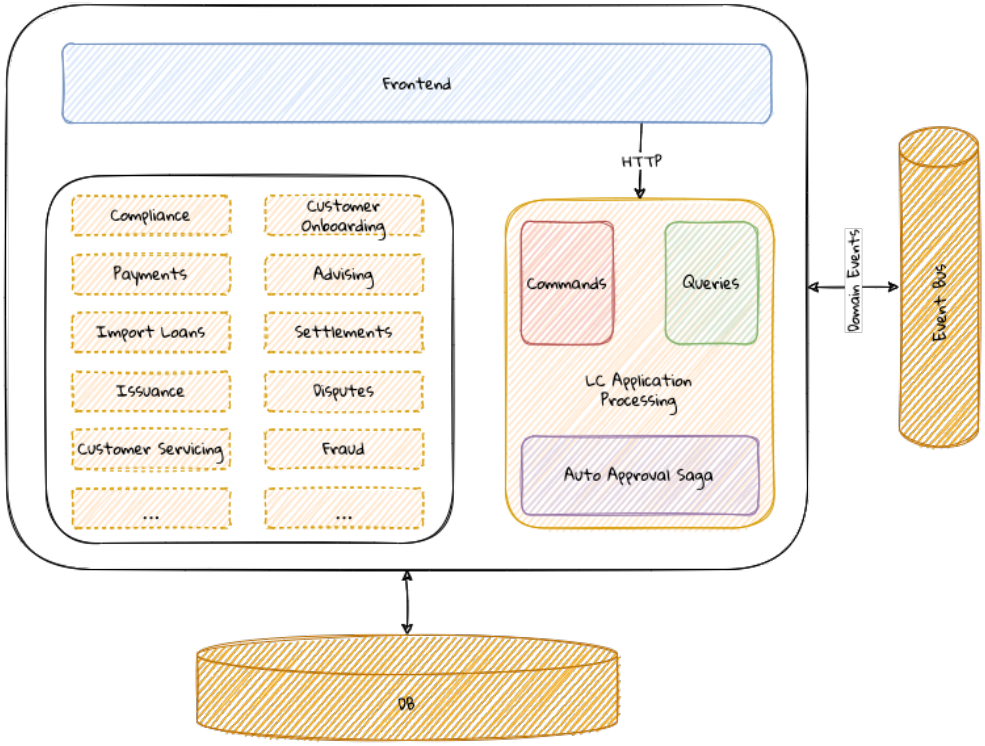


Figure 3. Out of process event bus introduced

Event bus options

TODO: Discuss event bus options

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:

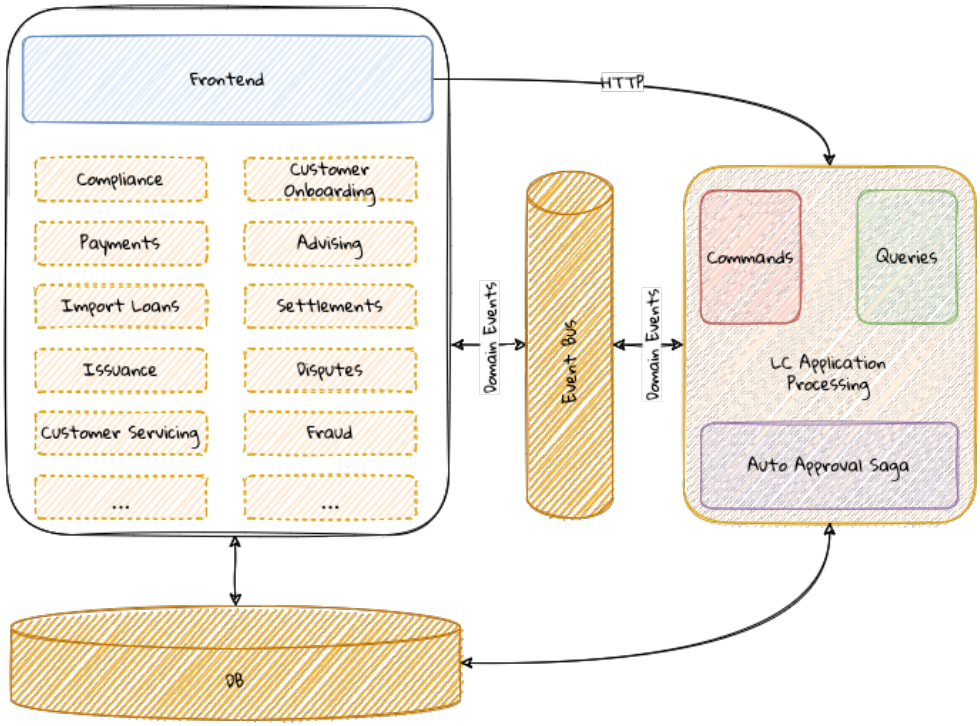


Figure 4. LC Application processing deployed independently

Database interactions

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

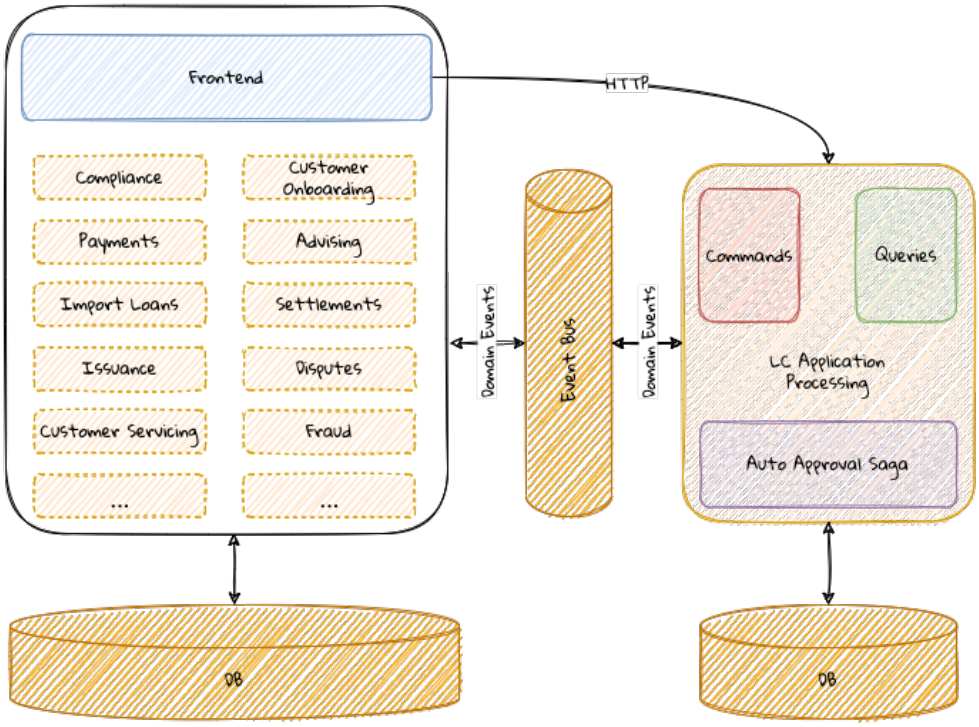
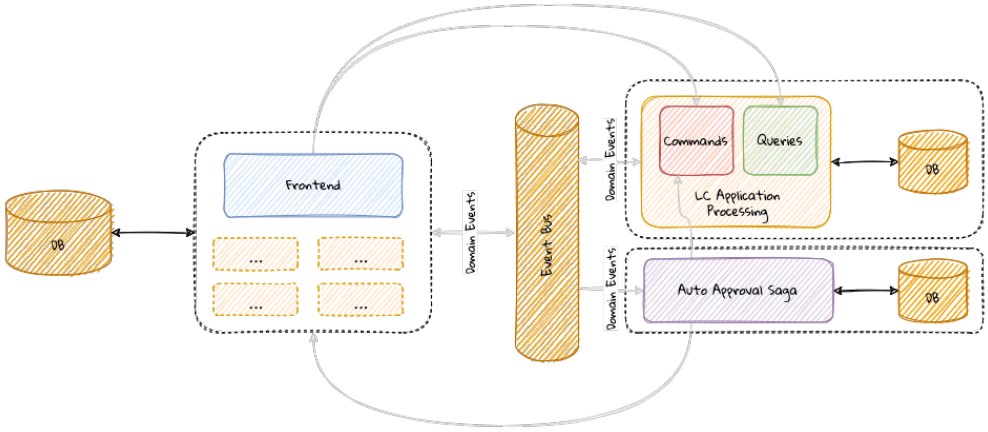


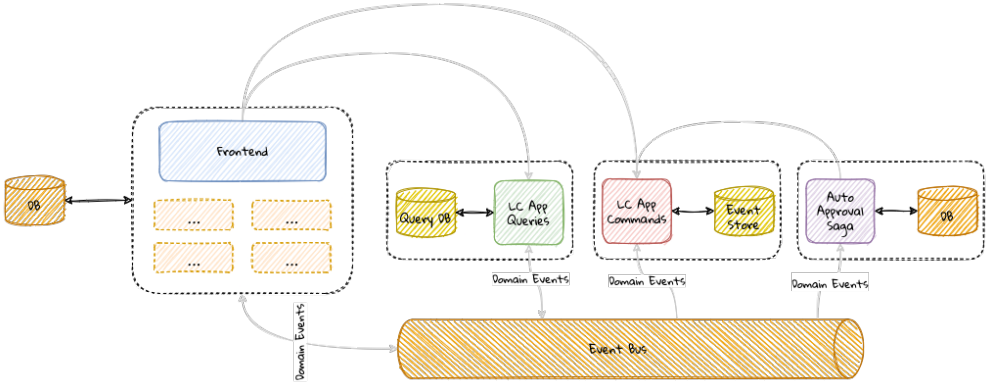
Figure 5. Independent data persistence

Potential next steps

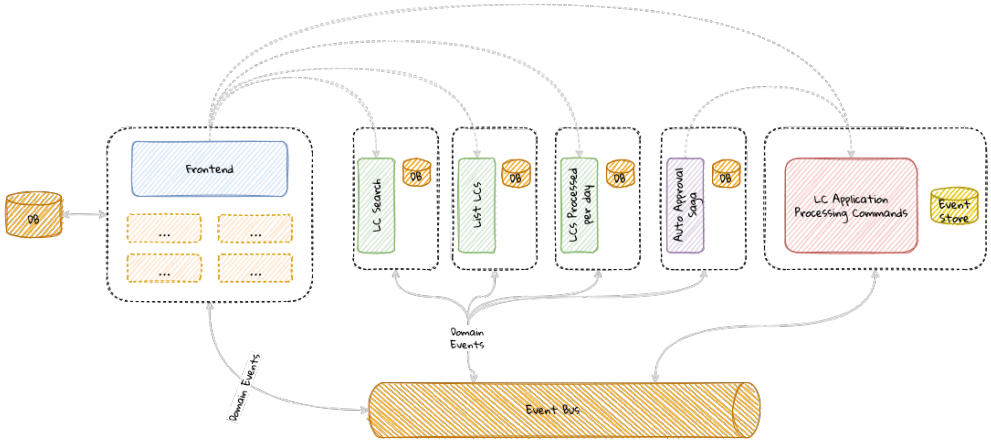
Sagas as standalone components



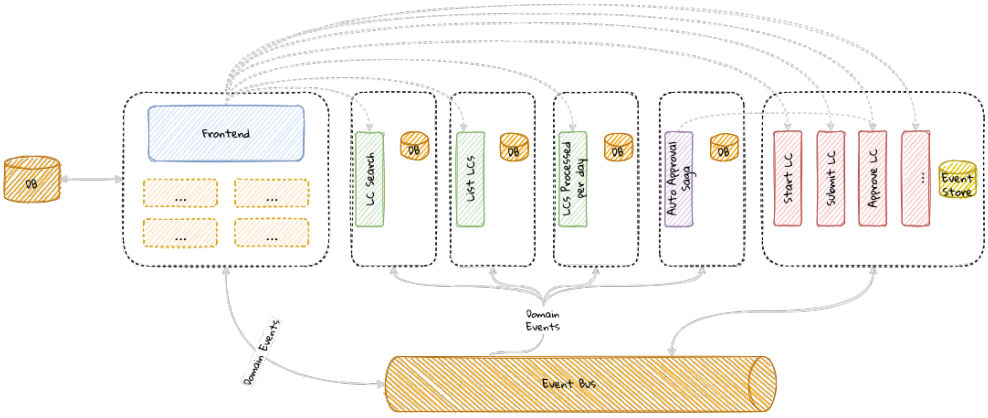
Commands and queries as standalone components



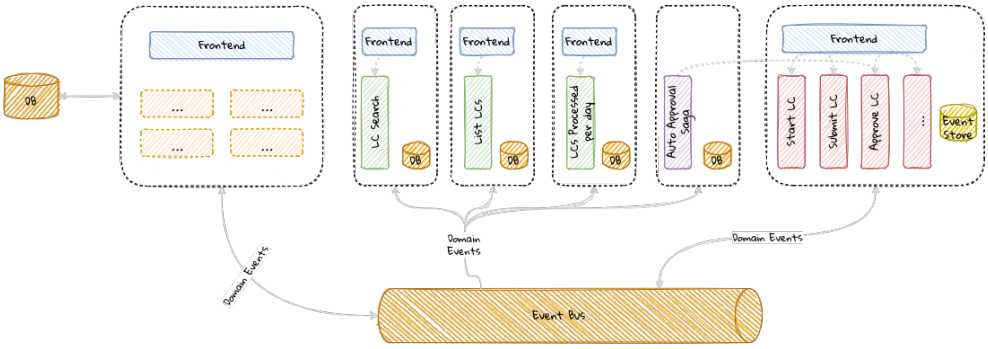
Distributing individual query components



Even more fine-grained distribution



Customer experiences and frontends



Decomposing and distributing big balls of mud

Code-first

Data-first

Non-technical implications of distribution

Team topologies

Tech stack

Technical implications of distribution

Performance

Dual writes

Transactional outbox

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

Testing strategies

Not yet finalized

Transitional architecture

Understanding the costs of distribution

Handling exceptions

Testing the Overall System

Compatibility