

Table of Contents

Integrating with External Systems	1
Technical Requirements	1
Continuing our design journey	1
Bounded context relationships	2
Symmetric relationship patterns	3
Asymmetric relationship patterns	6
Implementation patterns	10
Data-based	11
Code-based	12
API-based	13
Enforcing contracts	13
Further reading	13

Integrating with External Systems

Wholeness is not achieved by cutting off a portion of one's being, but by integration of the contraries.

— Carl Jung

Thus far, we have used DDD to implement a robust core for our application. However, most bounded contexts usually have both upstream and downstream dependencies which usually change at a pace which is different from these core components. To maintain both agility and reliability and enable loose coupling, it is important to create what DDD calls the anti-corruption layer in order to shield the core from everything that surrounds it. In this chapter, we will look at integrating with a legacy Inventory Management system. We will round off by looking at common patterns when integrating with legacy applications.

Technical Requirements

Continuing our design journey

From our domain analysis in the earlier chapters, we have arrived at four bounded contexts for our application as depicted here:



Figure 1. Context map for the LC solution

Thus far, our focus has been on the implementation of the internals of the **LC Application** bounded context. While the LC Application bounded context is independent of the other bounded contexts, it is not completely isolated from them. For example, when processing an LC application, we need to perform merchandise and applicant checks which require interactions with the **Compliance** and **Customer Onboarding** bounded contexts respectively. This means that these bounded contexts have a relationship with each other. These relationships are driven by the nature of collaboration between the teams working on the respective bounded contexts. Let's examine how these team dynamics influence integration mechanisms between bounded contexts in a way that continues to preserve their individual integrity.

Bounded context relationships

We need bounded contexts to as independent as possible. However, this does not mean that bounded contexts are completely isolated from each other. Bounded contexts need to collaborate with others to provide business value. Whenever there is collaboration required between two bounded contexts, the nature of their relationship is not only influenced by their individual goals and priorities, but also by the prevailing organizational realities. In a high performing environment, it is fairly common to have a single team assume ownership of a bounded context. The relationships between the teams owning these bounded contexts, play a significant role in influencing the integration patterns employed to arrive at a solution. At a high level, there are two

categories of relationships:

1. Symmetric
2. Asymmetric

Let's look at these relationship types in more detail.

Symmetric relationship patterns

Two teams can be said to have a symmetric relationship when they have an equal amount of influence in the decision-making process to arrive at a solution. Both teams are in a position to and indeed, do contribute more or less equally towards the outcome.



Figure 2. Both teams have an equal say in influencing the solution

There are three variations of symmetric relationships, each of which we outline in more detail here:

Partnership

In a partnership, both teams integrate in an adhoc manner. There are no fixed responsibilities assigned when needing complete integration work. Each team picks up work as and when needed without the need for any specific ceremony or fanfare. The nature of the integration is usually two-way with both teams exchanging solution artifacts as and when needed. Such relationships require extremely high degrees of collaboration and understanding of the work done by both teams.

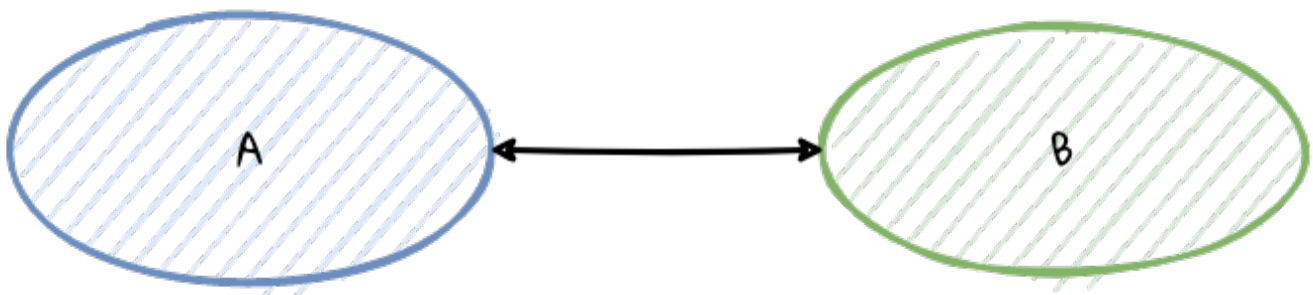


Figure 3. There is an adhoc, mutual dependency between teams in a partnership relationship

Example

A web front-end team working in close collaboration with the APIs team building the BFFs for the front-end. The BFF team creates experience APIs meant to be used exclusively by the front-end. To fulfill any functionality, the front-end team requires capabilities to be exposed by the APIs team. On the other hand, the APIs team is dependent on the front-end team to provide advice on what capabilities to build and the order in which to build them. Both teams freely make use of each

other's domain models (for example, the same set of request and response objects that define the API) to implement functionality. Such reuse happens mostly arbitrarily and when API changes happen, the both teams coordinate changes to keep things working.

When to use

Partnership relationship between teams require high levels of collaboration, trust and understanding. Teams tend to use this when team boundaries are informal. It also helps if these teams are co-located and/or have a significant working time overlap.

Potential pitfalls

Partnership relationships between teams can lead to a situation where individual team responsibilities become very unclear leading the solution towards the dreaded *big ball of mud*.

Shared kernel

Unlike in a partnership, when using a shared kernel, teams have a clear understanding of the solution artifacts and models they choose to share between themselves. Both teams take equal responsibility in the upkeep of these shared artifacts.

Example

The *LC Application Processing* and *Customer Onboarding* teams in our LC application may choose to use a common model to represent the `CustomerCreditValidatedEvent`. Any enhancements or changes to the event schema can affect both teams. The responsibility to make any changes is owned by both teams. Intentionally, these teams do not share anything beyond this mutually agreed upon models and artifacts.

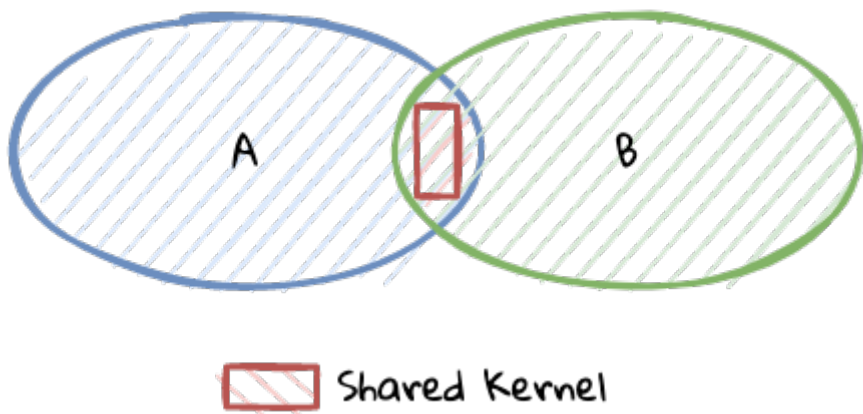


Figure 4. Teams have an explicit understanding of shared models

When to use

The shared kernel form of collaboration works well if shared artifacts are required to be consumed in an identical fashion in both contexts. Furthermore, it is attractive for multiple teams to coordinate and continue sharing, as opposed to duplicating identical models in both contexts.

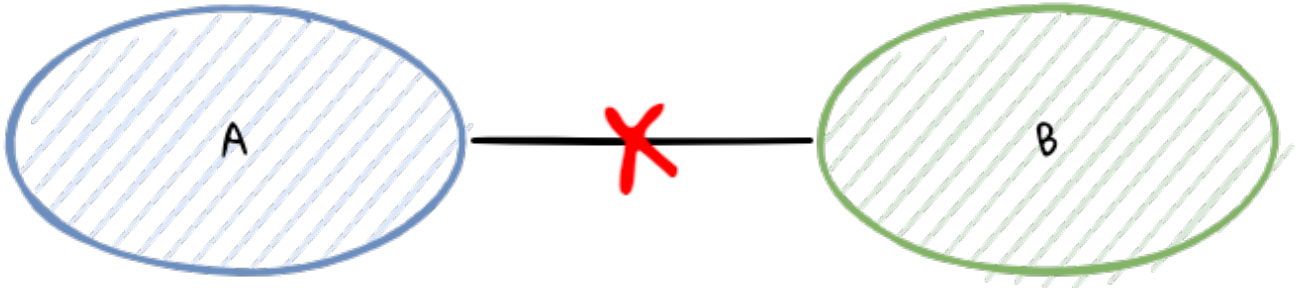
Potential pitfalls

Changes made to the shared kernel affect both bounded contexts. This means that any change made

to the shared kernel needs to remain compatible for both teams. Needless to say, as the number of teams using the shared kernel increases, the cost of coordination goes up manifold.

Separate ways

When two teams choose to not share any artifacts or models between them, they go their own separate ways.



Example

The *LC Application Processing* and the *Customer Onboarding* teams may start with sharing the same build/deployment scripts for their services. Over a period of time, deployment requirements may diverge to a point where the shared cost of maintaining these scripts becomes prohibitively expensive, causing these teams to fork their deployments to regain independence from the other team.

When to use

In some cases, two teams may be unable to collaborate for a variety of reasons, ranging from a drift in individual team requirements to organizational politics. Whatever the case may be, these teams may decide that the cost of collaboration is too high, resulting in them going their own separate ways.

Potential pitfalls

Choosing to go separate ways may result in duplicate work across affected bounded contexts. When working in bounded contexts that map to the core sub-domains, this may prove counter-productive as it could lead to inconsistent behaviors unintentionally.

It is possible to transition from one relationship type to another over a period of time. In our experience, transitioning from any one of these relationships may not be straightforward. In cases where requirements are relatively clear at the outset, it may be easier to start with a *shared kernel*. On the contrary, if requirements are unclear, it may be prudent to start either with a loose *partnership* or go *separate ways* until requirements become clear. In any of these scenarios, it is important to keep evaluating the nature of the relationship and transition to a more appropriate type based on our enhanced understanding of the requirements and/or the relationship itself.

In each of the relationships characterized above, the teams involved have a more or less equal say in how the relationship evolved and the resulting outcomes. However, this may not always be the case. Let's look at examples of cases where one team may have a clear upper hand in terms of how the relationship evolves.

Asymmetric relationship patterns

Two teams can be said to have an asymmetric relationship when one of the teams has a stronger influence in the decision-making process to arrive at a solution. In other words, there is a clear customer-supplier (or upstream-downstream) relationship where either the customer or the supplier plays a dominant role that affects solution design approaches. It is also likely that the customer and the supplier do not share common goals.

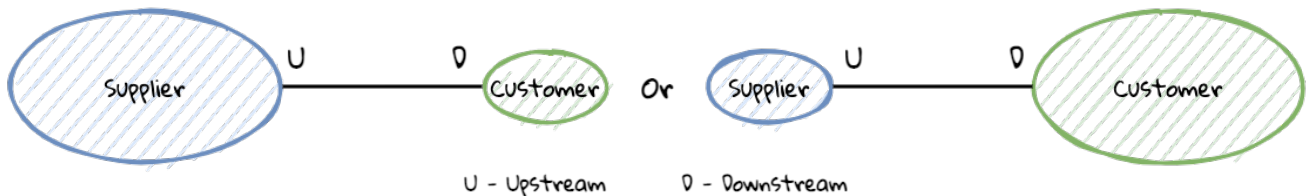


Figure 5. One of the teams has a dominant say in influencing the solution

There are at least three solution patterns when teams are in an asymmetric relationship, each of which we outline in more detail here:

Conformist (CF)

It is not unusual for the side playing the supplier role to have a dominant say in how the relationship with one or more customers is implemented. Furthermore, the customer may simply choose to conform with the supplier-provided solution as is, making it an integral part of their own solution. In other words, the supplier provides a set of models and the customer uses those same models to build their solution. In this case, the customer is termed to be a *conformist*.

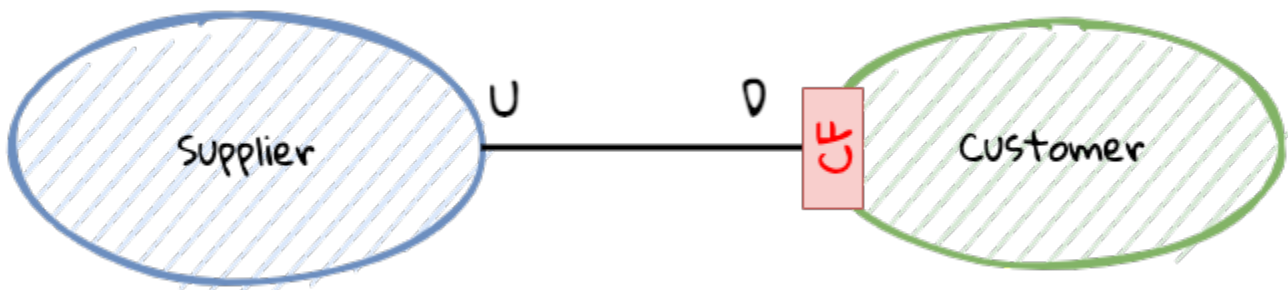


Figure 6. Customer accepts dependency on supplier model

Example

When building a solution to validate United States postal addresses of LC applicants, we chose to conform to the [USPS Web Tools](#) address validation API schema. Given that the business started with just US-based applicants, this made sense. This means that any references to the address model in our bounded contexts mimic the schema prescribed by the USPS. This further means that we will need to keep up with changes that occur in the USPS API as and when they occur (regardless of whether that change is needed for our own functionality).

When to use

Being a conformist is not necessarily a negative thing. The supplier's models may be a well accepted industry standard, or they may simply be good enough for our needs. It may also be that the team may not have the necessary skills, motivation or immediate needs to do something different from

what the supplier has provided. This approach also enables teams to make quick progress, leveraging work mostly done by other experts.

Potential pitfalls

An overuse of the conformist pattern may dilute the ubiquitous language of our own bounded contexts, resulting in a situation where there is no clear separation between the supplier and customer concepts. It may also be that concepts that are core to the supplier's context leaks into our own, despite those concepts carrying little to no meaning in our context. This may result in these bounded contexts being very tightly coupled with each other. And if a need arises to switch to another supplier or support multiple suppliers, the cost of change may be prohibitively expensive.

Anti-corruption layer (ACL)

There may be scenarios where a customer may need to collaborate with the supplier, but may want to shield itself from the supplier's ubiquitous language and models. In such cases, it may be prudent to redefine these conflicting models in the customer's own ubiquitous language using a translation layer at the time of integration, also known as an *anti-corruption layer* (ACL).

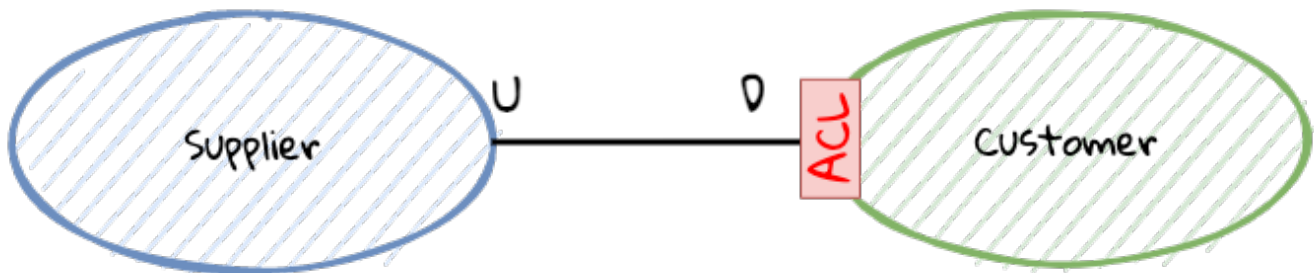
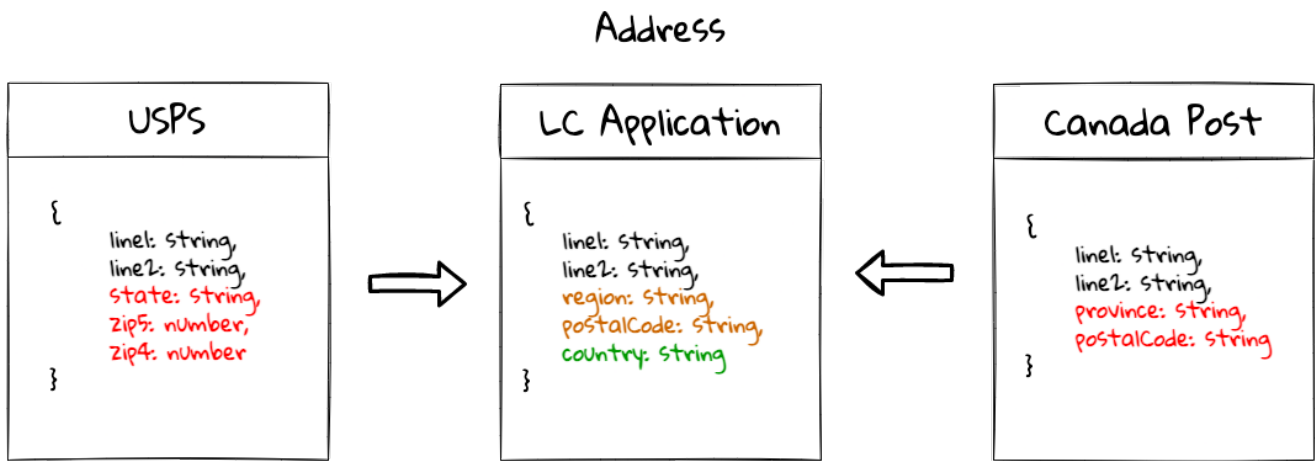


Figure 7. Customer wants to protect itself from supplier models

Example

In the address validation example referenced in the [Conformist](#) section, the *LC Application Processing* team may need to support Canadian applicants as well. In such a case, being a conformist to a system that supports only US addresses may prove restrictive and even confusing. For example, the US *state* is analogous to a *province* in Canada. Similarly, *zip code* in the US is referred to *postal code* in Canada. In addition, US zip codes are numeric whereas Canadian postal codes are alphanumeric. Most importantly, we currently do not have the notion of a *country code* in our address model, but now we will need to introduce this concept to differentiate addresses within the respective countries. Let's look at the address models from the respective countries here:



While we initially conformed to the USPS model, we have now evolved to support more countries. For example, *region* is used to represent the concept of *state/province*. Also, we have introduced the *country* value object, which was missing earlier.

When to use

Anti-corruption layers come in handy when the customer models are part of a core domain. The ACL shields the customer from changes in the supplier's models and can help produce more loosely coupled integrations. It may also be necessary when we are looking to integrate similar concepts from multiple suppliers.

Potential pitfalls

Using an anti-corruption layer may be tempting in a lot of cases. However, it is less beneficial when the concepts being integrated don't often change, or are defined by a well known authority. Using an ACL with a custom language may only cause more confusion. Creating an ACL usually requires additional translations and thereby may increase the overall complexity of the customer's bounded context and may be considered premature optimization.

Open host service (OHS)

Unlike the conformist and the anti-corruption layer, where customers do not have a formal means to interface with the supplier, with the open host service, the supplier defines a clear interface to interact with its customers. This interface may be made available in the form of a well-known published language (for example, a REST interface or a client SDK):

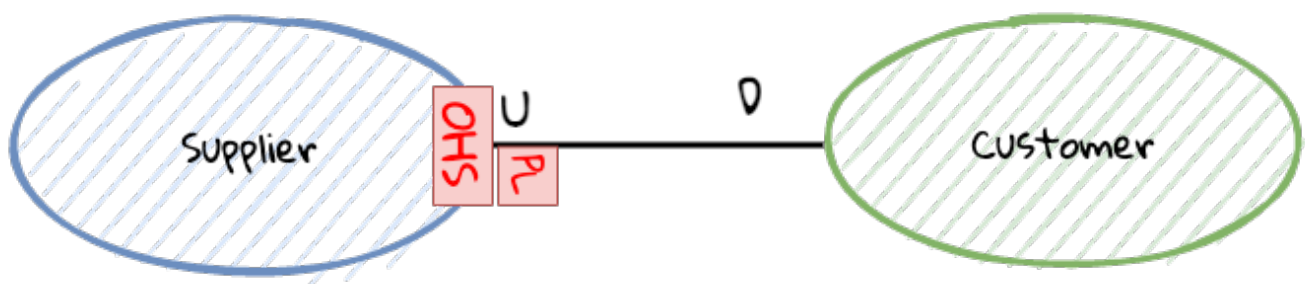


Figure 8. Open host service (OHS) using a published language (PL).

Example

The LC Application Processing bounded context can expose an HTTP interface for each of its commands as shown here:

```
# 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'

# Change the amount on an existing application
curl POST /applications/ac130002/change-amount \
  -d '{"amount": 100, "currency": "USD"}' \
  -H 'content-type:application/vnd.lc-application.v2+json'

# Other commands omitted for brevity
```

As an augment to the HTTP interface shown here, we may even provide a client SDK in some of the more popular languages used by our customers. This helps hide more implementation details such as the MIME type and version from customers.

When to use

When the supplier wants to hide its internal models (ubiquitous language), making an open host service enables the supplier to evolve while providing a stable interface to its customers. In a sense, the open-host service pattern is a reversal of the anti-corruption layer pattern: instead of the customer, the supplier implements the translation of its internal model. Also, the supplier can consider providing an open host service when it is interested in providing a richer user experience for its customers.

Potential pitfalls

While suppliers may have good intentions by providing an open host service for its customers, it may result in increased implementation complexity (for example, there may be a need to support multiple versions of an API, or client SDKs in multiple languages). If the open host service does not take into account common usage patterns of its customers, it may result in a poor customer usability and also in degraded performance for the supplier.

It is important to note that the conformist and the anti-corruption layer are patterns that customers implement, whereas the open host service is a supplier-side pattern. For example, the following scenario with the supplier providing an *open host service* and one customer is a *conformist* while another has an *anti-corruption layer*, can be true as depicted here:



Figure 9. Asymmetric relationships with multiple customers.

Now that we have seen the various ways in which bounded contexts can integrate with each other, here is one possible implementation for our LC application depicted in the form of a context map:



Figure 10. Simplified context map for the LC application.

Thus far we have examined the various ways in which inter-team dynamics influence integration mechanisms. While having clarity at the conceptual level helps, let's see how these relationships manifest themselves at the implementation level.

Implementation patterns

We have looked integration between bounded contexts at a design level, but these concepts need to be translated into code. There are three broad categories that can be employed when integrating two bounded contexts:

1. Data-based
2. Code-based
3. API-based

Let's look at each method in more detail now.

Data-based

In this style of integration, the bounded contexts in question share data between each other. If the relationship is symmetric, the teams owning these bounded contexts may choose to share entire databases with free access to both read, write and change underlying structures. Whereas in an asymmetric relationship, the supplier may constrain the scope of access, based on the type of relationship.

Shared database

The simplest form of data integration is the use of a shared database. In this style of integration, all participating bounded contexts have unrestricted access to the schemas and the underlying data as shown here:

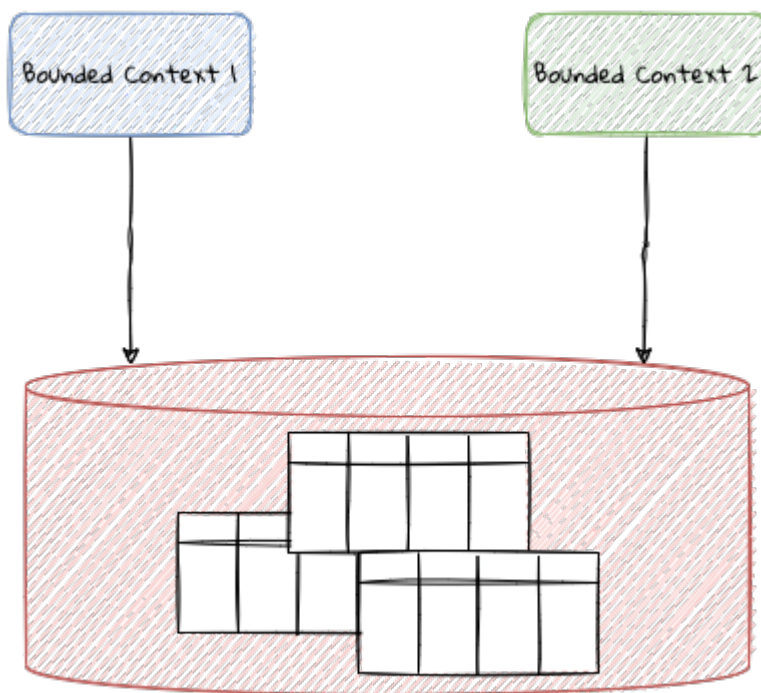


Figure 11. Integration using a shared database

When to use

The shared database presents a very low barrier to entry for teams looking to quickly enable new or enhance existing functionality by providing ready access to data for read and/or write use-cases. More importantly, it also allows the use of local database transactions, which usually provides strong consistency, lower complexity and better performance (especially when working with relational databases).

Potential pitfalls

However, this symmetric integration style where multiple teams have shared ownership is usually frowned upon because it often leads to a situation where there is no clear ownership. Furthermore, the shared databases can become a source of tight coupling, accelerating the path towards the dreaded *big ball of mud*. Additionally, users of the shared database can suffer from the *noisy neighbor* effect where one co-tenant monopolizing resources adversely affects all other tenants. For these reasons, teams will be well advised to choose this style of integration sparingly.

Replicated data

In the case of asymmetric relationships, suppliers may be unwilling to provide direct access to their data. However, they may choose to integrate with customers using a mechanism based on data sharing. An alternate form of integration is to provide a copy of the data required by consumers. There are many variations on how this can be implemented, we depict the more common ways here:



Figure 12. Integration using data replication.

- **Database views:** In this form, the consumer gets or is provided access to a subset of the data using query-based or materialized views. In either case, the customer usually has read-only access to the data and both supplier and customer continue to share the same physical resources (usually the DB engine).
- **Full read replica:** In this form, the customer gets access to a read replica of the supplier's entire database, usually on physically disparate infrastructure.
- **Partial read replica:** In this form, the customer gets access to a read replica of a subset of the supplier's database, again on physically disparate infrastructure.

When to use

This style of integration may be required when there is an asymmetric relationship between the supplier and the customer. Like the shared database, this integration style usually requires less upfront effort to integrate. This is also apt when suppliers intend to provide read-only access to a subset of their data. It may also suffice to use data replication when customers only require to read a subset of the supplier's data.

Potential pitfalls

If we choose to use database views, we may continue to suffer from the noisy neighbor effect. On the other hand, if we choose to create physically disparate replicas, we will need to incur the cost of additional operational complexity. More importantly, the consumers remain tightly coupled to the supplier's domain models and ubiquitous language.

Code-based

Code sharing

Code libraries

Stored procedures

API-based

Request-response APIs

Publish-subscribe APIs

Enforcing contracts

Further reading

Title	Author	Location
Integration database	Martin Fowler	https://martinfowler.com/bliki/IntegrationDatabase.html