stances, any more than microservices are—but it's a choice nonetheless. If we fall into the trap of systematically denigrating the monolith as a viable option for delivering our software, then we're at risk of not doing right by ourselves or the users of our software. We'll further explore the trade-offs around monoliths and microservices in Chapter 3, and discuss some tools that will help you better assess what is right for your own context.

## On Coupling and Cohesion

Understanding the balancing forces between coupling and cohesion is important when defining microservice boundaries. *Coupling* speaks to how changing one thing requires a change in another; *cohesion* talks to how we group related code. These concepts are directly linked. Constantine's law articulates this relationship well:

> A structure is stable if cohesion is high, and coupling is low.
>
> —Larry Constantine

This seems like a sensible and useful observation. If we have two pieces of tightly related code, cohesion is low as the related functionality is spread across both. We also have tight coupling, as when this related code changes, both things need to change.

If the structure of our code system is changing, that will be expensive to deal with, as the cost of change across service boundaries in distributed systems is so high. Having to make changes across one or more independently deployable services, perhaps dealing with the impact of breaking changes for service contracts, is likely to be a huge drag.

The problem with the monolith is that all too often it is the opposite of both. Rather than tend toward cohesion, and keep things together that tend to change together, we acquire and stick together all sorts of unrelated code. Likewise, loose coupling doesn't really exist: if I want to make a change to a line of code, I may be able to do that easily enough, but I cannot deploy that change without potentially impacting much of the rest of the monolith, and I'll certainly have to redeploy the entire system.

We also want system stability because our goal, where possible, is to embrace the concept of independent deployability—that is, we'd like to be able to make a change to our service and deploy that service into production *without having to change anything else*. For this to work, we need stability of the services we consume, and we need to provide a stable contract to those services that consume us.

Given the wealth of information out there about these terms, it would be silly of me to revisit things too much here, but I think a summary is in order, especially to place these ideas in the context of microservice architectures. Ultimately, these concepts of cohesion and coupling influence hugely how we think about microservice architec-

ture. And this is no surprise—cohesion and coupling are concerns regarding modular software, and what is microservice architecture other than modules that communicate via networks and can be independently deployed?

### A Brief History of Coupling and Cohesion

The concepts of cohesion and coupling have been around in computing for a long time, with the concepts initially outlined by Larry Constantine in 1968. These twin ideas of coupling and cohesion went on to form the basis of much of how we think about writing computer programs. Books like *Structured Design* by Larry Constantine & Edward Yourdon (Prentice Hall, 1979) subsequently influenced generations of programmers subsequently (this was required reading for my own university degree, almost 20 years after it was first published).

Larry first outlined his concepts of cohesion and coupling in 1968 (an especially auspicious year for computing) at the National Symposium on Modular Programming, the same conference where Conway's law first got its name. That year also gave us two now infamous NATO-sponsored conferences during which software engineering as a concept also rose to prominence (a term previously coined by Margaret H. Hamilton).

## Cohesion

One of the most succinct definitions I've heard for describing cohesion is this: "the code that changes together, stays together." For our purposes, this is a pretty good definition. As we've already discussed, we're optimizing our microservice architecture around ease of making changes in business functionality—so we want the functionality grouped in such a way that we can make changes in as few places as possible.

If I want to change how invoice approval is managed, I don't want to have to hunt down the functionality that needs changing across multiple services, and then coordinate the release of those newly changed services in order to roll out our new functionality. Instead, I want to make sure the change involves modifications to as few services as possible to keep the cost of change low.

## Coupling

> Information Hiding, like dieting, is somewhat more easily described than done.
>
> —David Parnas, *The Secret History Of Information Hiding*

We like cohesion we like, but we're wary of coupling. The more things are "coupled", the more they have to change together. But there are different types of coupling, and each type may require different solutions.

There has been a lot of prior art when it comes to categorizing types of coupling, notably work done by Meyer, Yourdan, and Constantine. I present my own, not to say that the work done previously is wrong, more than I find this categorization more useful when helping people understand aspects associated to the coupling of distributed systems. As such, it isn't intended to be an exhaustive classification of the different forms of coupling.

### Information Hiding

A concept that comes back again and again when it comes to discussions around coupling is the technique called *information hiding*. This concept, first outlined by David Parnas in 1971, came out of his work looking into how to define module boundaries.[6]

The core idea with information hiding is to separate the parts of the code that change frequently from the ones that are static. We want the module boundary to be stable, and it should hide those parts of the module implementation that we expect to change more often. The idea is that internal changes can be made safely as long as module compatibility is maintained.

Personally, I adopt the approach of exposing as little as possible from a module (or microservice) boundary. Once something becomes part of a module interface, it's hard to walk that back. But if you hide it now, you can always decide to share it later.

Encapsulation as a concept in object-oriented (OO) software is related, but depending on whose definition you accept may not be quite the same thing. Encapsulation in OO programming has come to mean the bundling together of one or more things into a container—think of a class containing both fields and the methods that act on those fields. You could then use visibility at the class definition to hide parts of the implementation.

For a longer exploration of the history of information hiding, I recommend Parnas's "The Secret History of Information Hiding."[7]

#### Implementation coupling

*Implementation coupling* is typically the most pernicious form of coupling I see, but luckily for us it's often one of the easiest to reduce. With implementation coupling, A

6 Although Parnas's well known 1972 paper "On the Criteria to be Used in Decomposing Systems into Modules" is often cited as the source, he first shared this concept in "Information Distributions Aspects of Design Methodology", Proceedings of IFIP Congress '71, 1971.

7 See Parnas, David, "The Secret History of Information Hiding." Published in *Software Pioneers*, eds. M. Broy and E. Denert (Berlin Heidelberg: Springer, 2002).

is coupled to B in terms of how B is implemented—when the implementation of B changes, A also changes.

The issue here is that implementation detail is often an arbitrary choice by developers. There are many ways to solve a problem; we choose one, but we may change our minds. When we decide to change our minds, we don't want this to break consumers (independent deployability, remember?).

A classic and common example of implementation coupling comes in the form of sharing a database. In Figure 1-9, our Order service contains a record of all orders placed in our system. The Recommendation service suggests records to our customers that they might like to buy based on previous purchases. Currently, the Recommendation service directly accesses this data from the database.

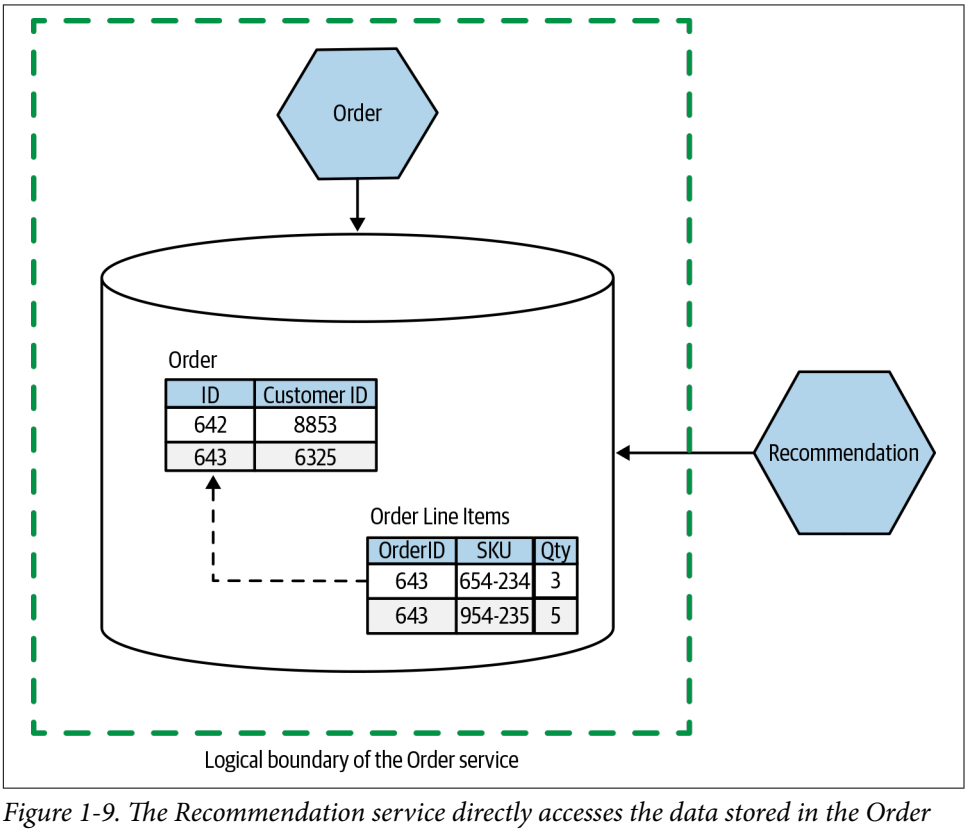


*Figure 1-9. The Recommendation service directly accesses the data stored in the Order service*

Recommendations require information about which orders have been placed. To an extent, this is unavoidable *domain* coupling, which we'll touch on in a moment. But in this particular situation, we are coupled to a specific schema structure, SQL dialect,

and perhaps even the content of the rows. If the Order service changes the name of a column, splits the Customer Order table apart, or whatever else, it conceptually still contains order information, but we break how the Recommendation service fetches this information. A better choice is to hide this implementation detail, as Figure 1-10 shows—now the Recommendation service accesses the information it needs via an API call.

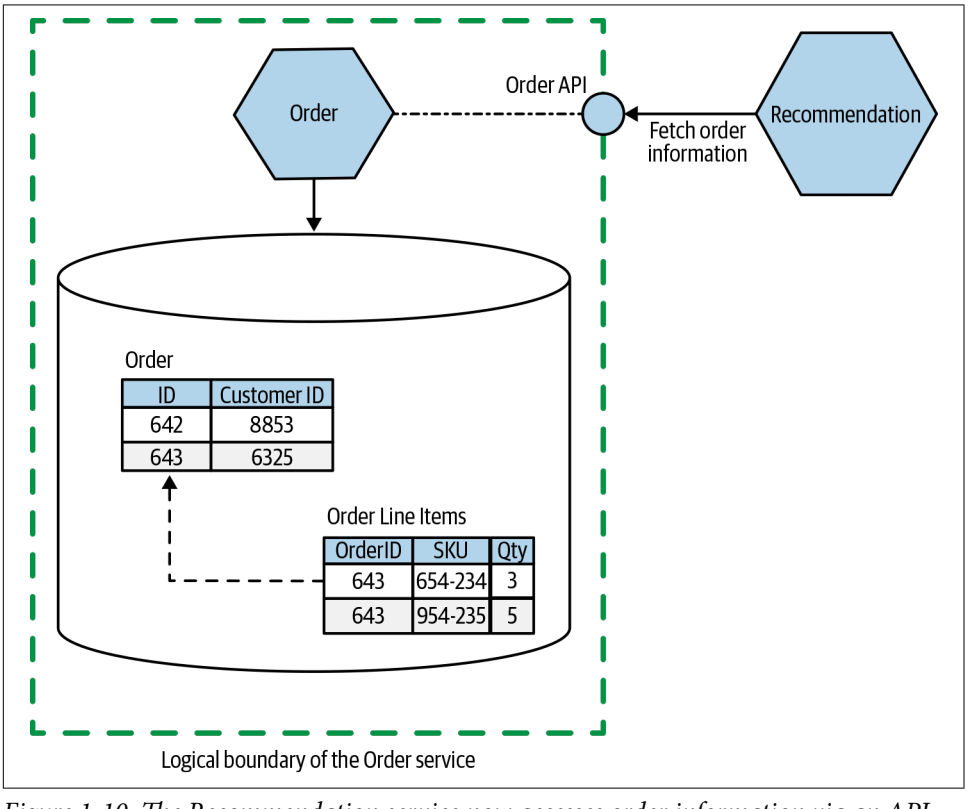


*Figure 1-10. The Recommendation service now accesses order information via an API, hiding internal implementation detail*

We could also have the Order service publish a dataset, in the form of a database, which is meant to be used for bulk access by consumers—just as we see in Figure 1-11. As long as the Order service can publish data accordingly, any changes made inside the Order service are invisible to consumers, as it maintains the public contract. This also opens up the opportunity to improve the data model exposed for consumers, tuning to their needs. We'll be exploring patterns like this in more detail in Chapters 3 and 4.

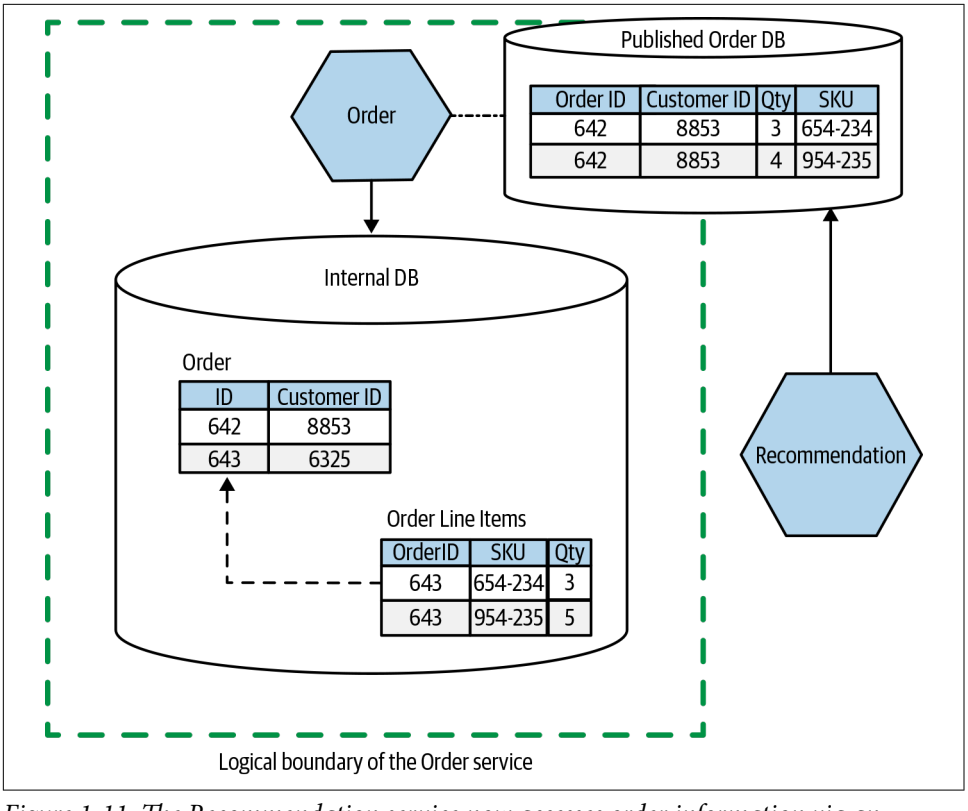


*Figure 1-11. The Recommendation service now accesses order information via an exposed database, which is structured differently from the internal database*

In effect, with both of the preceding examples, we are making use of information hiding. The act of hiding a database behind a well-defined service interface allows the service to limit the scope of what is exposed, and can allow us to change how this data is represented.

Another helpful trick is to use "outside-in" thinking when it comes to defining a service interface—drive the service interface by thinking of things from the point of the service consumers first, and then work out how to implement that service contract. The alternative approach (which I have observed is all too common, unfortunately) is to do the reverse. The team working on the service takes a data model, or another internal implementation detail, then thinks to expose that to the outside world.

With "outside-in" thinking, you instead first ask, "What do my service consumers need?" And I don't mean you ask *yourself* what your consumers need; I mean you actually ask the people that will call your service!

Treat the service interfaces that your microservice exposes like a user interface. Use outside-in thinking to shape the interface design in partnership with the people who will call your service.

Think of your service contract with the outside world as a user interface. When designing a user interface, you ask the users what they want, and iterate on the design of this with your users. You should shape your service contract in the same way. Aside from the fact it means you end up with a service that is easier for your consumers to use, it also helps keep some separation between the external contract and the internal implementation.

### Temporal coupling

*Temporal coupling* is primarily a runtime concern that generally speaks to one of the key challenges of synchronous calls in a distributed environment. When a message is sent, and how that message is handled is connected in time, we are said to have temporal coupling. That sounds a little odd, so let's take a look at an explicit example in Figure 1-12.

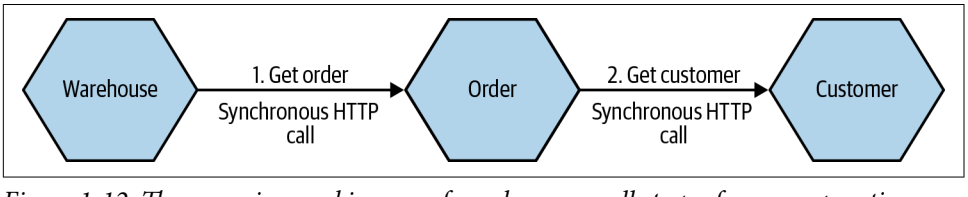


*Figure 1-12. Three services making use of synchronous calls to perform an operation can be said to be temporally coupled*

Here we see a synchronous HTTP call made from our Warehouse service to a downstream Order service to fetch required information about an order. To satisfy the request, the Order service in turn has to fetch information from the Customer service, again via a synchronous HTTP call. For this overall operation to complete, the Warehouse, Order, and Customer services all needed to be up, and contactable. They are temporally coupled.

We could reduce this problem in various ways. We could consider the use of caching —if the Order service cached the information it needed from the Customer service, then the Order service would be able to avoid temporal coupling on the downstream service in some cases. We could also consider the use of an asynchronous transport to send the requests, perhaps using something like a message broker. This would allow a message to be sent to a downstream service, and for that message to be handled after the downstream service is available.

A full exploration of the types of service-to-service communication is outside the scope of this book, but is covered in more detail in Chapter 4 of *Building Microservices*.

### Deployment coupling

Consider a single process, which consists of multiple statically linked modules. A change is made to a single line of code in one of the modules, and we want to deploy that change. In order to do that, we have to deploy the entire monolith—even including those modules that are unchanged. Everything must be deployed together, so we have *deployment coupling*.

Deployment coupling may be enforced, as in the example of our statically linked process, but can also be a matter of choice, driven by practices like a release train. With a release train, preplanned release schedules are drawn up in advance, typically with a repeating schedule. When the release is due, all changes made since the last release train gets deployed. For some people, the release train can be a useful technique, but I strongly prefer to see it as a transitional step toward proper release-on-demand techniques, rather than viewing it as an ultimate goal. I even have worked in organizations that would deploy all services in a system all at once as part of these release train processes, without any thought to whether those services need to be changed.

Deploying something carries risk. There are lots of ways to reduce the risk of deployment, and one of those ways is to change only what needs to be changed. If we can reduce deployment coupling, perhaps through decomposing larger processes into independently deployable microservices, we can reduce the risk of each deployment by reducing the scope of deployment.

Smaller releases make for less risk. There is less to go wrong. If something does go wrong, working out what went wrong and how to fix it is easier because we changed less. Finding ways to reduce the size of release goes to the heart of continuous delivery, which espouses the importance of fast feedback and release-on-demand methods.[8] The smaller the scope of the release, the easier and safer it is to roll out, and the faster feedback we'll get. My own interest in microservices comes from a previous focus on continuous delivery—I was looking for architectures that made adoption of continuous delivery easier.

Reducing deployment coupling doesn't require microservices, of course. Runtimes like Erlang allow for the hot-deployment of new versions of modules into a running

8 See Jez Humble and David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Upper Saddle River: Addison Wesley, 2010) for more details.

process. Eventually, perhaps more of us may have access to such capabilities in the technology stacks we use day to day.[9]

### Domain coupling

Fundamentally, in a system that consists of multiple independent services, there has to be some interaction between the participants. In a microservice architecture, *domain coupling* is the result—the interactions between services model the interactions in our real domain. If you want to place an order, you need to know what items were in a customer's shopping basket. If you want to ship a product, you need to know where you ship it. In our microservice architecture, by definition this information may be contained in different services.

To give a concrete example, consider Music Corp. We have a warehouse that stores goods. When customers place orders for CDs, the folks working in the warehouse need to understand what items need to be picked and packaged, and where the package needs to be sent. So, information about the order needs to be shared with the people working in the warehouse.

Figure 1-13 shows an example of this: an Order Processing service sends all the details of the order to the Warehouse service, which then triggers the item to be packaged up. As part of this operation, the Warehouse service uses the customer ID to fetch information about the customer from the separate Customer service so that we know how to notify them when the order is sent out.

In this situation, we are sharing the entire order with the warehouse, which may not make sense—the warehouse needs only information about what to package and where to send it. They don't need to know how much the item cost (if they need to include an invoice with the package, this could be passed along as a pre-rendered PDF). We'd also have problems with information that we have to control access to being too widely shared—if we shared the full order, we could end up exposing credit card details to services that don't need it, for example.

9 Greenspun's 10th rule states, "Any sufficiently complicated C or Fortran program contains an ad hoc, informally specified, bug-ridden, slow implementation of half of Common Lisp." This has morphed into a newer joke: "Every microservice architecture contains a half-broken reimplementation of Erlang." I think there is a lot of truth to this.

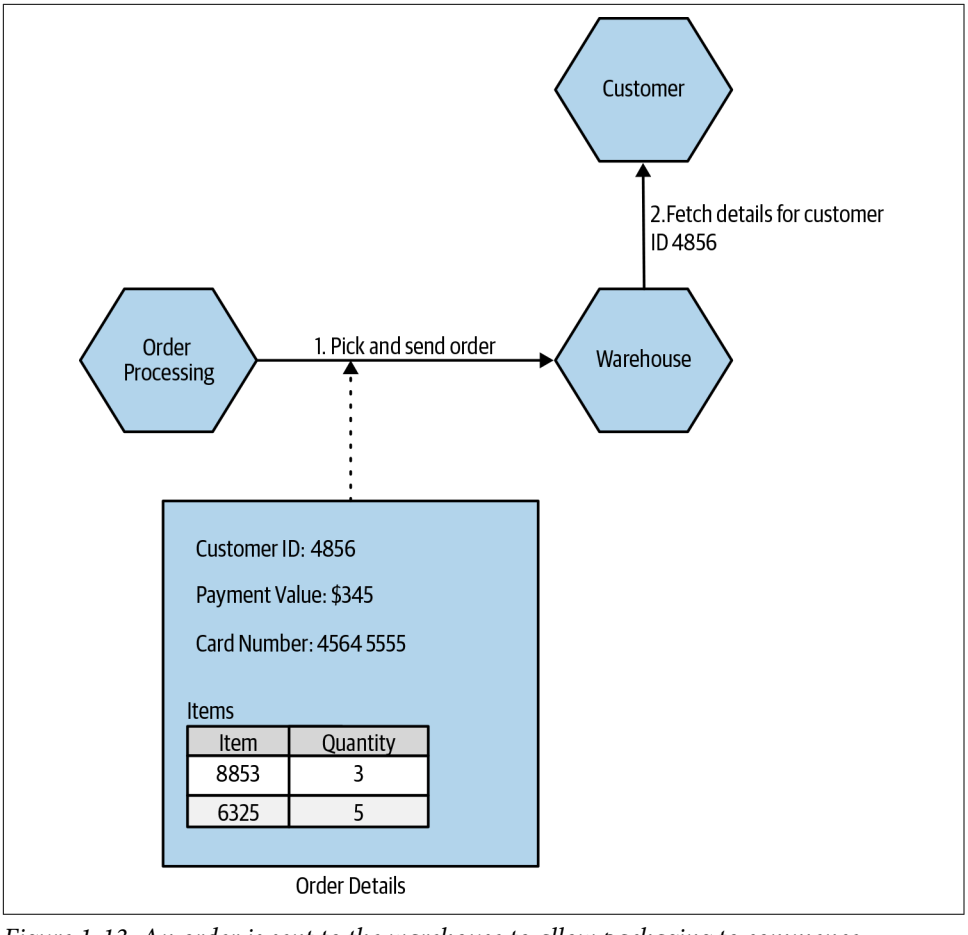


*Figure 1-13. An order is sent to the warehouse to allow packaging to commence*

So instead, we might come up with a new domain concept of a Pick Instruction containing just the information the Warehouse service needs, as we see in Figure 1-14. This is another example of information hiding.

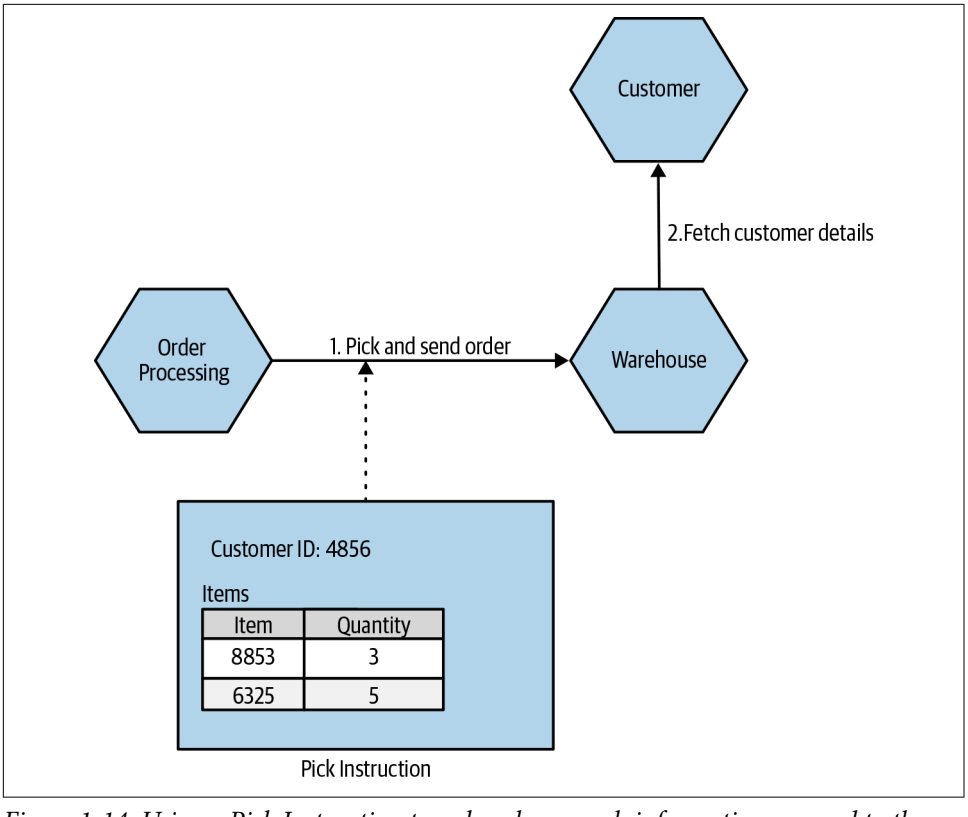


*Figure 1-14. Using a Pick Instruction to reduce how much information we send to the Warehouse service*

We could further reduce coupling by removing the need for the Warehouse service to even need to know about a customer if we wanted to—we could instead provide all appropriate details via the Pick Instruction, as Figure 1-15 shows.

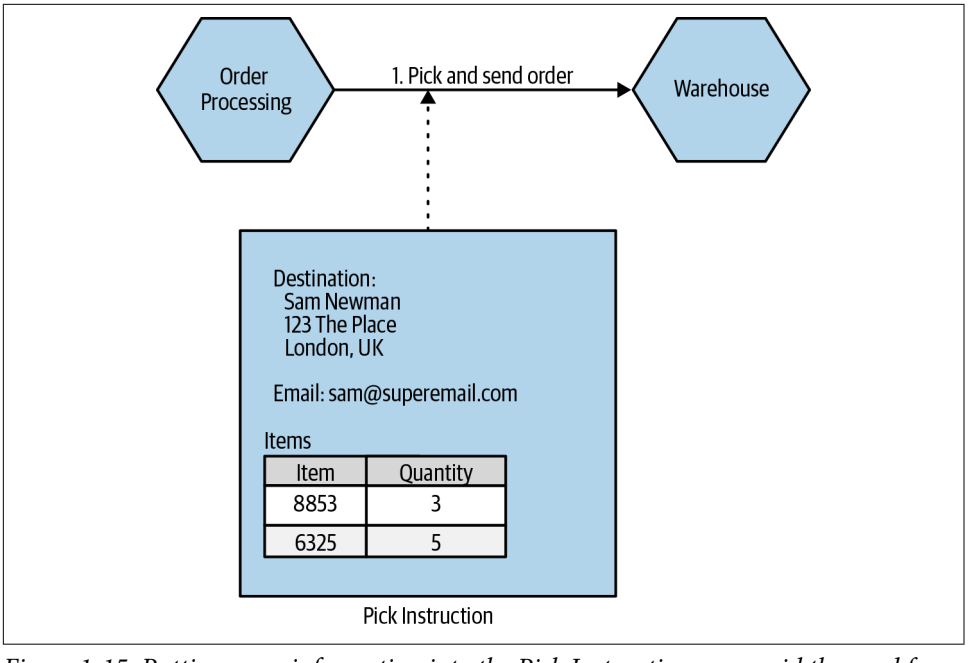


*Figure 1-15. Putting more information into the Pick Instruction can avoid the need for a call to the Customer service*

For this approach to work, it probably means that at some point Order Processing has to access the Customer service to be able to generate this Pick Instruction in the first place, but it's likely that Order Processing would need to access customer information for other reasons anyway, so this is unlikely to be much of an issue. This process of "sending" a Pick Instruction implies an API call being made from Order Processing to the Warehouse service.

An alternative could be to have Order Processing emit some kind of event that the Warehouse consumes, in Figure 1-16. By emitting an event that the Warehouse consumes, we effectively flip the dependencies. We go from Order Processing depending on the Warehouse service to be able to ensure an order gets sent, to the Warehouse listening to events from the Order Processing service. Both approaches have their merits, and which I would choose would likely depend on a wider understanding of the interactions between the Order Processing logic and the functionality encapsulated in the Warehouse service—that's something that some domain modeling can help with, a topic we'll explore next.

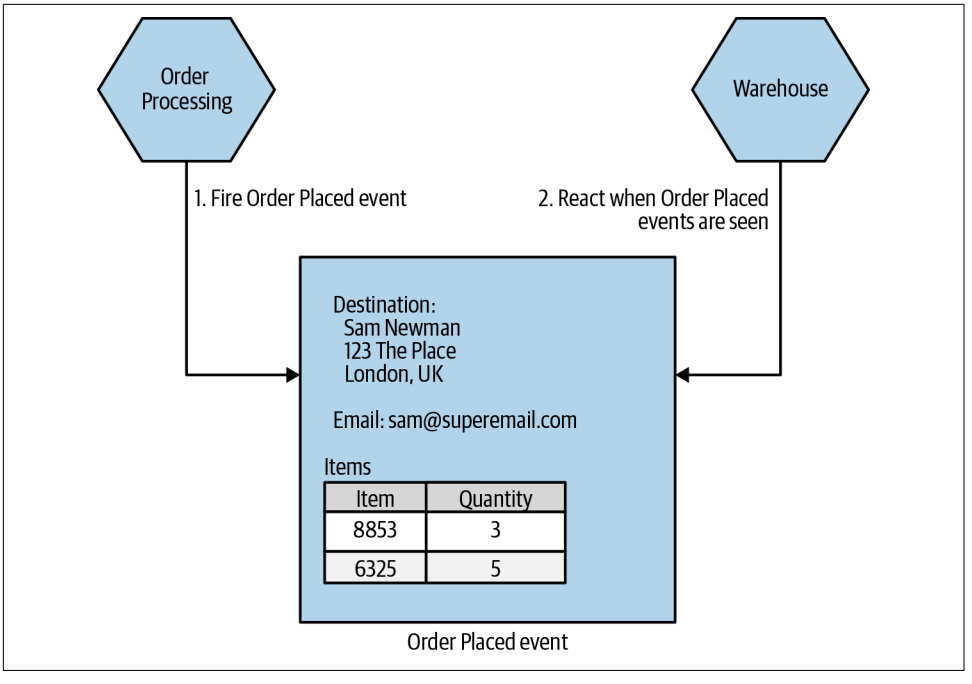


*Figure 1-16. Firing an event that the Warehouse service can receive, containing just enough information for the order to be packaged and sent*

Fundamentally, some information is needed about an order for the Warehouse service to do any work. We can't avoid that level of domain coupling. But by thinking carefully about what and how we share these concepts, we can still aim to reduce the level of coupling being used.

## Just Enough Domain-Driven Design

As we've already discussed, modeling our services around a business domain has significant advantages for our microservice architecture. The question is how to come up with that model—and this is where domain-driven design (DDD) comes in.

The desire to have our programs better represent the real world in which the programs themselves will operate is not a new idea. Object-oriented programming languages like Simula were developed to allow us to model real domains. But it takes more than program language capabilities for this idea to really take shape.