Service Design

Microservice Boundaries

So just how micro should a microservice be? In reality, there is no simple answer for this question. The things that first come to mind, such as lines of code in a microservice or the size of a team working on one are compelling, since they offer the chance to focus on a *quantifiable* value. However, the problem with these measures is that they ignore the business context of what we are implementing. They don't address the organizational context of who is implementing the service and, more importantly, how the service is being used within your system.

Instead of trying to find some quantity to measure, we find most companies focus on a *quality* of each microservice—the use case or context in which the component will be used. Many microservice adopters have turned to Eric Evans' "domain-driven design" (DDD) approach for a well-established set of processes and practices that facilitate effective, business-context-friendly modularization of large complex systems.

Microservice Boundaries and Domain-Driven Design

Essentially, what we see people doing when they introduce the microservices way into their companies is that they begin to decompose existing components into smaller parts in order to increase their ability to improve the quality of the service faster without sacrificing reliability.

There are many ways to decompose a large system into smaller subsystems. In one case we may be tempted to decompose a system based on implementation technology.

For instance, we can say that all computationally heavy services need to be written in C or Rust and therefore they are a separate subsystem, while I/O-heavy features could certainly benefit from the nonblocking I/O of a technology such as Node.js and therefore they are a subsystem of their own. Alternatively, we can divide a large system based on team geography: one subsystem may be written in the US, while others may be developed and maintained by software teams in Africa, Asia, Australia, Europe, or South America. Intuitively, giving a self-contained subsystem for development to a team that is located in one place is well- optimized. Another reason you may decide to divide a system based on geography is that specific legal, commercial, and cultural requirements of operating in a particular market may be better understood by a local team. Can a software development team from New York accurately capture all the necessary details of an accounting software that will be used in Cairo?

Eric Evans (book *Domain-Driven Design*) outlines a fresh approach to determining boundaries of subsystems in the context of a larger system. In the process, he offers a model-centric view of software system design. They provide an abstract way to look at something—a way that highlights the things we are interested in. Evans notes that most large systems don't actually have a single model. The overall model of a large system is actually comprised of many smaller models that are intermingled with each other. These smaller models are organic representations of relevant business contexts—they make sense in their context and when used within the context they are intuitive for a person who is the subject matter expert of the context.

Bounded Context

In DDD, Evans points out that teams need to be very careful when combining contextual models to form a larger software system. He puts it this way:

> Multiple models are in play on any large project. Yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should not be applied.

The microservices way attempts to break large components (models) into smaller ones in order to reduce the confusion and bring more clarity to each element of the system. As such, microservice architecture is an architectural style that is highly compatible with the DDD way of modeling. Each component in the system lives within its own bounded context, which means the model for each component and these context models are only used within their bounded scope and are not shared across the bounded contexts.

It is generally acknowledged that properly identifying bounded contexts in a system, using DDD techniques, and breaking up a large system along the seams of those bounded contexts is an effective way of designing microservice boundaries.

Sam Newman (book Building Microservices) makes an important point here: bounded contexts represent autonomous business domains (i.e., distinct business capabilities), and therefore are the appropriate starting point for identifying the dividing lines for microservices. If we use the DDD and bounded contexts approaches, the chances of two microservices needing to share a model and the corresponding data space, or ending up having tight coupling, are much lower. Avoiding data sharing improves our ability to treat each microservice as an independently deployable unit. And independent deployability is how we can increase our speed while still maintaining safety within the overall system. Using DDD and bounded contexts is an excellent process for designing components.

Smaller Is Better

The notion of work-unit granularity is a crucial one in many contexts of modern software development. Whether defined explicitly or implicitly, we can clearly see the trend showing up in such foundational methodologies as Agile Development, Lean Startup, and Continuous Delivery, among others. These methodologies have revolutionized project management, product development, and DevOps, respectively.

It is interesting to note that each one of them has the principle of size reduction at its core: reducing the size or scope of the problem, reducing the time it takes to complete a task, reducing the time it takes to get feedback, and reducing the size of the deployment unit. These all fall into a notion we call “batch-size reduction"

For example, moving to Agile from Waterfall can be viewed as a reduction of the “batch size” of a development cycle—if the cycle was taking many months in Waterfall, now we strive to complete a similar batch of tasks: define, architect, design, develop, and deploy, in much shorter cycles (weeks versus months). Granted, the Agile Manifesto lists other important principles as well, but they only reinforce and complement the core principle of “shorter cycles” (i.e., reduced batch size).

Once you adopt the notion of limited batch size from Agile, Lean, and Continuous Delivery at the code, project, and deployment level, it makes sense to think about applying it at the architecture level as well. So, in the simplest terms, this “limited batch size” is the “micro” in microservice.

Ubiquitous Language

Here’s what one of the prominent authorities in the space of DDD, Vaughn Vernon, had to say about the optimal size of a bounded context:

> *Bounded context should be as big as it needs to be in order to fully express its complete ubiquitous language.*

In DDD, we need a shared understanding and way of expressing the domain specifics. This shared understanding should provide business and tech teams with a common language that they can use to collaborate on the definition and implementation of a model. Just as DDD tells us to use one model within a component (the bounded context),

the language used within that bounded context should be coherent and pervasive—what we in DDD call ubiquitous language.

From a purely technical perspective, the smaller the microservice the easier it can be developed quicker (Agile), iterated on quicker (Lean), and deployed more frequently (Continuous Delivery). But on the modeling side, it is important to avoid creating services that are "too small." According to Vernon, we cannot arbitrarily reduce the size of a bounded context because its optimal size is determined by the business context (model). Our technical need for the size of a service can sometimes be different (smaller) from what DDD modeling can facilitate. Bounded contexts are a great start, but we need more tools in our toolbelt if we are to size microservices efficiently.

API Design for Microservices

When considering microservice component boundaries, the source code itself is only part of our concern. Microservice components only become valuable when they can communicate with other components in the system. They each have an interface or API.

We see two practices in crafting APIs for microservices worth mentioning here:

- Message-oriented
- Hypermedia-driven

Message-Oriented

Just as we work to write component code that can be safely refactored over time, we need to apply the same efforts to the shared interfaces between components. The most effective way to do this is to adopt a message-oriented implementation for microservice APIs. The notion of messaging as a way to share information between components dates back to the initial ideas about how object-oriented programming would work.

All of the companies we talked with about microservice component design mentioned the notion of messaging as a key design practice. For example, Netflix relies on message formats like Avro, Protobuf, and Thrift over TCP/IP for communicating internally and JSON over HTTP for communicating to external consumers (e.g., mobile phones, browsers, etc.). By adopting a message-oriented approach, developers can expose general entry points into a component (e.g., an IP address and port number) and receive task-specific messages at the same time. This allows for changes in message content as a way of refactoring components safely over time. The key lesson learned here is that for far too long, developers have viewed APIs and web services as tools to transmit serialized "objects" over the wire. However, a more efficient approach is to look at a complex system as a collection of services exchanging messages over a wire.

Hypermedia-driven

In these instances, the messages passed between components contain more than just data. The messages also contain descriptions of possible actions (e.g., links and forms). Now, not just the data is loosely coupled—so are the actions. For example, Amazon's API Gateway and App- Stream APIs both support responses in the Hypertext Application Language (HAL) format.

Hypermedia-style APIs embrace evolvability and loose coupling as the core values of the design style. Regardless of the name used, if we are to design proper APIs in microservice architecture, it helps to get familiar with the hypermedia style.

Hypermedia style is essentially how HTML works for the browser. HTTP messages are sent to an IP address (your server or client location on the Internet) and a port number (usually "80" or "443"). The messages contain the data and actions encoded in HTML format.

The hypermedia API style is as transformative to the API space as object-oriented design was for code design. A long time ago, we used to just write endless lines of code (maybe lightly organizing them in functions), but then object-oriented design came by with a revolutionary idea: "what if we grouped the state and the methods that operate on that state in an autonomous unit called *an object*, thus encapsulating data and behavior?" In essence, hypermedia style has very similar approach but for API design. This is an API style in which API messages contain both data and *controls* (e.g., metadata, links, forms), thus dynamically guiding API clients by responding with not just static data but also control metadata describing API affordances.

Hypermedia APIs are more like the human Web: evolvable, adaptable, versioning- free—when was the last time you cared about what "version" of a website you are looking at? As such, hypermedia-style APIs are less brittle, more discoverable, and fit right at home in a highly distributed, collaborative architectural style such as microservices.

Data and Microservices

As software engineers, we have been trained to think in terms of data, first and foremost. To give the simplest example, it has pretty much been ingrained in our "muscle memory," or whatever the mental equivalent of one is, to start system design by first designing the pertinent data models. When asked to build an application, the very first task most software engineers will complete is identifying entities and designing database tables for data storage. This is an efficient way of designing centralized systems and whole generations of programmers have been trained to think this way. But data-centric design is not a good way to implement distributed systems—especially systems that rely on independently deployable microservices. The biggest reason for this is the absence of strong, centralized, uniform control over the entire system in the case of distributed systems, which makes a formerly efficient process inefficient.

Shipping, Inc.

Let's assume that we are designing a microservice architecture for a shipment company, Shipping Inc. As a parcel-delivery company, they need to accept packages, route them through various sorting warehouses (hops on the route), and eventually deliver to the destination. Shipping, Inc. is building native mobile applications for a variety of platforms to let customers track their packages all the way from pickup to final delivery. These mobile applications will get the data and functionality they need from a set of microservices.

Let's imagine that Shipping, Inc.'s accounting and sales subsystems (microservices) need access to daily currency exchange rates to perform their operations. A data- centric design would create a table or set of tables in a database that contain exchange rates. Then we would let various subsystems query our database to retrieve the data. This solution has significant issues—two microservices depend on the design of the shared table and data in it, leading to tight coupling and impeding independent deployability.

If instead, we had viewed "currency exchange rates" as a capability and had built an independent microservice (currency rates) serving the sales and accounting microservices, we would have had three independent services, all loosely coupled and independently deployable.

Furthermore, since, by their nature, APIs in services hide implementation details, we can completely change the data persistence supporting the currency rates service (e.g., from MySQL to Cassandra, if scalability became an

issue) without any of the service's consumers noticing the change or needing to adjust. Last but not least, since services (APIs) are able to put forward alternative interfaces to its various consumers, we can easily alter the interface that the currency rates microservice provides to the sales microservice, without affecting the accounting microservice, thus fulfilling the promise of independent evolution, a necessity for independent deployability. Mission accomplished.

Let's see what techniques we can use to avoid data-sharing in complex use cases.

- *event sourcing*
- CQRS—command query responsibility segregation.

Event Sourcing

One of the most widespread of those habits is structural data modeling. It has become very natural for us to describe models as collections of interacting logical entities and then to map those logical entities to physical tables where the data is stored. More recently, we have started using NoSQL and object stores that take us slightly away from the relational world, but in essence the approach is still the same: we design structural entities that model objects around us and then we "save" the object's state in a database store of some kind. Whether storage happens in table rows and columns, serialized as JSON strings, or as object graphs, we are still performing CRUD-based modeling. But this is not the only way to model the world. Instead of storing structures that model the state of our world, we can store events that lead to the current state of our world. This modeling approach is called event sourcing.

> Event sourcing is all about storing facts and any time you have "state" (structural models)—they are first-level derivative off of your facts. And they are transient.

In this context, by "facts" Young means the representative value of an event occurrence. An example could be "a package was transported from the last sorting facility, out for final delivery."

Event sourcing is not some bleeding-edge, untested theory dreamed up to solve problems in microservices. Event sourcing has been used in the financial industry with great success, independent of any microservice architecture association.

In addition, the roots and inspiration for event sourcing go way beyond microservices, the Internet itself, or even computers—all the way back to financial accounting and the paper-and-pen ledgers that contain a list of transactions, and never just the end value ("state") of a balance. Think of your bank account: there's a balance amount for your checking and savings accounts, but those are not first-class values that banks store in their databases. The account balance is always a derivative value; it's a function. More specifically, the balance is the sum of all transactions from the day you opened your account.

If you do find any errors with any of the transactions, the bank will issue a "compensating transaction" to fix the error. This is another crucial property of event sourcing. In event sourcing, data is immutable—we always issue a new command/event to compensate rather than update a state of an entity, as we would do in a CRUD style.

When event sourcing is introduced to developers, the immediate concern is usually performance. If any state value is a function of events, we may assume that every access to the value would require recalculation of the current state from the source events. Obviously that would be extremely slow and generally unacceptable. Fortunately, in event sourcing, we can avoid such expensive operations by using a so-called rolling snapshot—a projection of the entity state at a given point in time. Depending on the event source implementation, it is common to snapshot intermediary values at various time points.

Shipping, Inc as a parcel-delivery company, they need to accept packages, route them through various sorting warehouses (hops on the route), and eventually deliver to their destinations.

A representative data model for this system executed in structural style is shown in below figure.

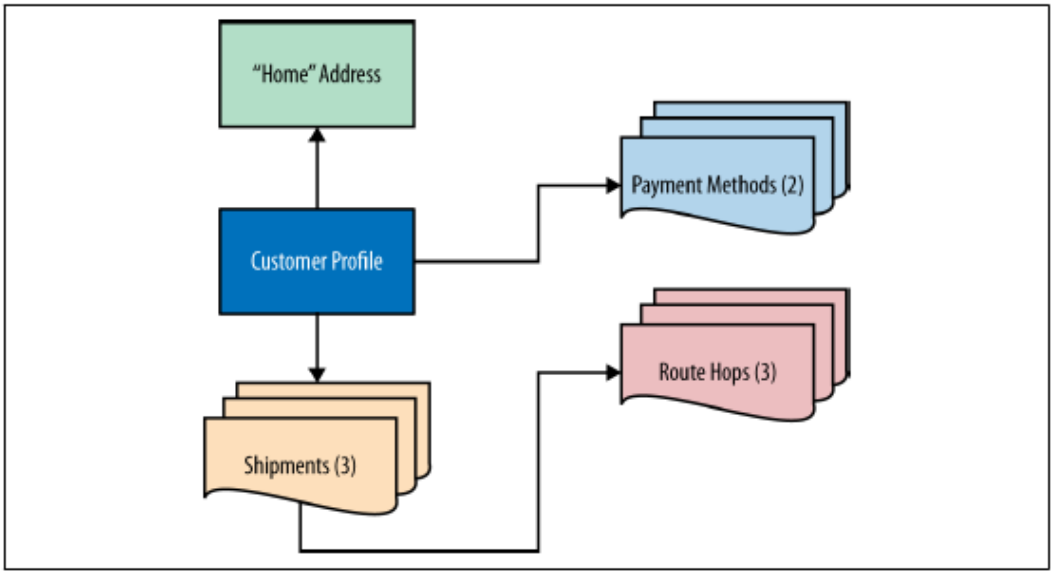

Figure 5-1. Data model for Shipping, Inc. using "current state" approach

The corresponding events-based model is shown in Figure 5-2.

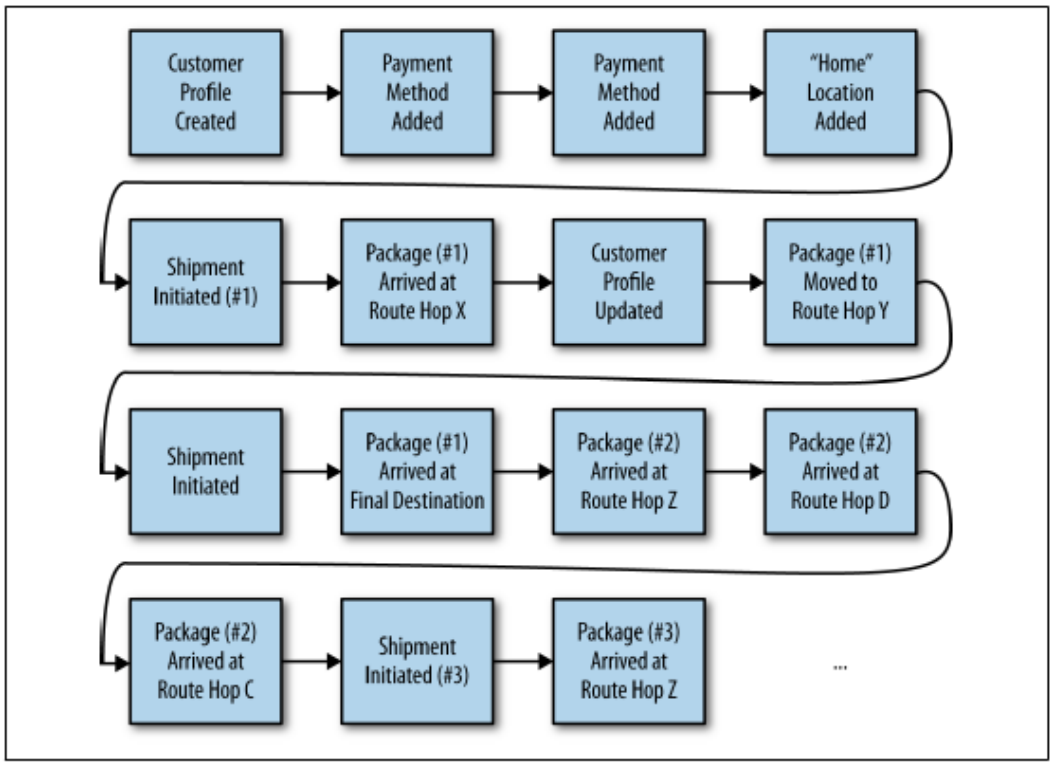

Figure 5-2. Data model for Shipping, Inc. using event sourcing

As you can see, the structural model strives to only save the current state of the system, while the event sourcing approach saves individual "facts" State, in event sourcing, is a function of all the pertinent facts that occurred.

System Model for Shipping, Inc.

As we noted earlier, a good start for a microservice system design is to identify bounded contexts in the system. Figure below shows a context map for key bounded contexts in our problem space.

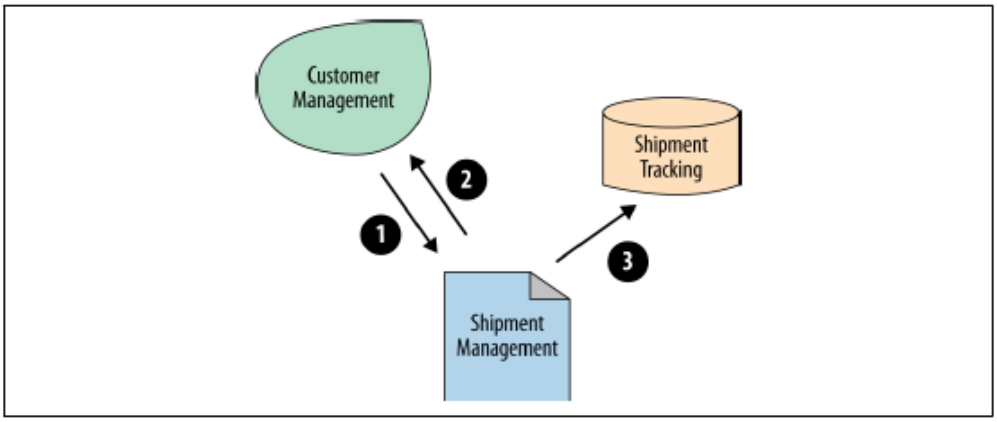

Figure 5-3. High-level context map for Shipping, Inc.'s microservice architecture

What are the capabilities of the three contexts and some of the data flows between the contexts, depicted by the arrows and numbers on the graph? They are as follows:

1. Customer Management creates, edits, enables/disables customer accounts, and can provide a representation of a customer to any interested context.
2. Shipment Management is responsible for the entire lifecycle of a package from drop-off to final delivery. It emits events as the package moves through sorting and forwarding facilities, along the delivery route.
3. Shipment Tracking is a reporting application that allows end users to track their shipments on their mobile device.

If we were to implement a data model of this application using a traditional, structural, CRUD-oriented model we would immediately run into data sharing and tight- coupling problems. Indeed, notice that the Shipment Management and Shipment Tracking contexts will have to query the same tables, at the very least the ones containing the transitions along the route. However, with event sourcing, the Shipment Management bounded context (and its corresponding microservice) can instead record events/commands and issue event notifications for other contexts and those other contexts will build their own data indexes (projections), never needing direct access to any data owned and managed by the Shipment Management microservice.

CQRS

Command query responsibility segregation is a design pattern that states that we can (and sometimes should) separate data-update versus data-querying capabilities into separate models. It tracks its ancestry back to a principle called command-query separation (CQS). CQRS instruct us to use entirely different models for updates versus queries.

Figure below shows a conceptual diagram that depicts CQRS for our Shipping, Inc. application.

As you can see, thanks to CQRS, we were able to completely separate the data models of the Shipment Management and Tracking microservices. In fact, Shipping Management doesn't even need to know about the existence of the Tracking microservice, and the only thing the Tracking microservice relies on is a stream of events to build its query index. During runtime the Tracking microservice only queries its own index.

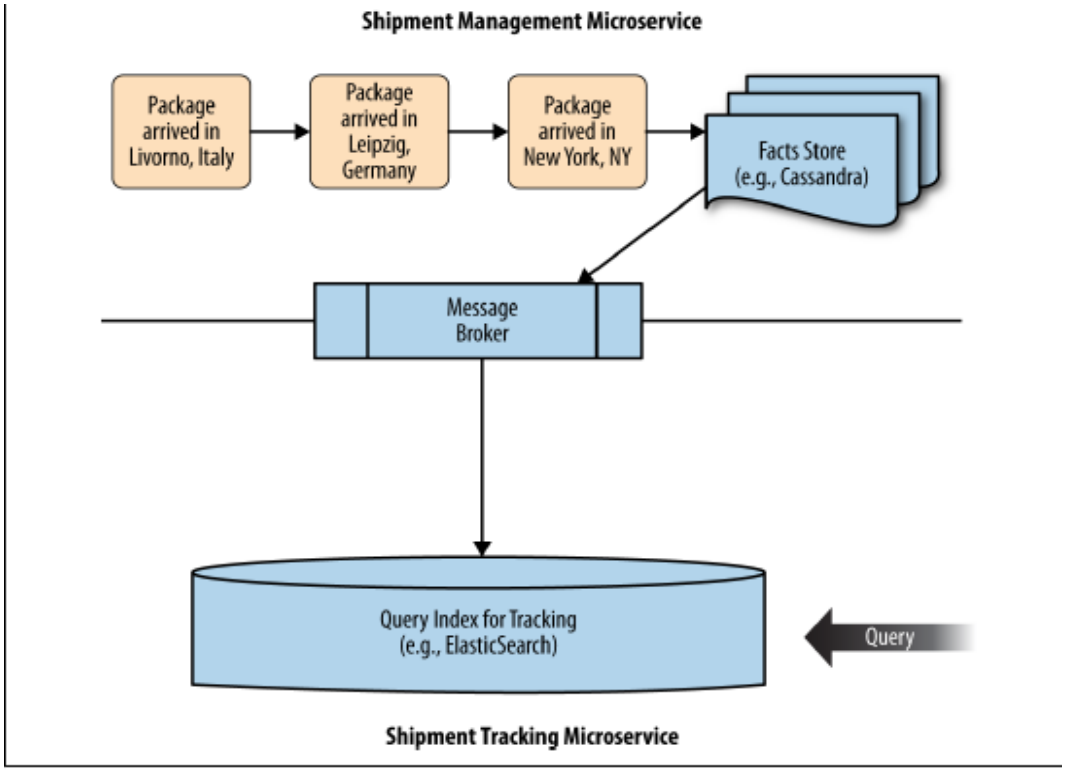

Figure 5-4. Data flow in command-query responsibility segregation (CQRS)-based model for Shipping, Inc.

The big win with using event sourcing and CQRS is that they allow us to design very granular, loosely coupled components. With bounded contexts our boundaries have to align with business capabilities and subdomain boundaries. With event sourcing, we can literally create microservices so tiny that they just manage one type of event or run a single report.

Distributed Transactions and Sagas

The shared data model is not the only use case that can introduce tight coupling between microservices. Another important threat is workflows. A lot of real-life processes cannot be represented with a single, atomic operation, since they are a sequence of steps. When we are dealing with such workflows, the result only makes sense if all of the steps can be executed. In other words, if any step in the sequence fails, the resulting state of the relevant system becomes invalid. You probably recognize this problem from RDBMS systems where we call such processes “transactions.”. Once the transaction is fully executed, we can remove the locks, or if any step of the transaction steps fails, we can roll back the steps already attempted.

For distributed workflows and share-nothing environments, we cannot use traditional transaction implementations with data locks and ACID compliance, since such transactions require shared data and local execution. Instead, an effective approach many teams use is known as “Sagas”. Sagas were designed for long-lived, distributed transactions.

Sagas are very powerful because they allow running transaction-like, reversible workflows in distributed, loosely coupled environments without making any assumptions on the reliability of each component of the complex system or the overall system itself. The compromise here is that Sagas cannot always be rolled back to the exact initial state of the system before the transaction attempt. But we can make a best effort to bring the system to a state that is consistent with the initial state through compensation.

In Sagas, every step in the workflow executes its portion of the work, registers a callback to a "compensating transaction" in a message called a "routing slip," and passes the updated message down the activity chain. If any step downstream fails, that step looks at the routing slip and invokes the most recent step's compensating transaction, passing back the routing slip. The previous step does the same thing, calling its predecessor compensating transaction and so on until all already executed transactions are compensated.

Consider this example: let's say a customer mailed a prepaid cashier's check for $100 via Shipping, Inc.'s insured delivery. When the courier showed up at the destination, they found out that the address was wrong and the resident wouldn't accept the package. Thus, Shipping, Inc. wasn't able to complete the transaction. Since the package was insured, it is Shipping, Inc.'s responsibility to "roll back" the transaction and return the money to the sender. With ACID-compliant transactions, Shipping, Inc. is supposed to bring the exact $100 check back to the original sender, restoring the system state to its exact initial value. Unfortunately, on the way back the package was lost. Since Shipping, Inc. could no longer "roll back" the transaction, they decided to reimburse the insured value of $100 by depositing that amount into the customer's account. Since this was an active, long-time Shipping, Inc. customer and a rational human being, they didn't care which $100 was returned to them. The system didn't return to its exact initial state, but the compensating transaction brought the environment back to a consistent state.

This is basically how Sagas work. Due to its highly fault-tolerant, distributed nature, Sagas are very well-suited to replace traditional transactions when transactions across microservice boundaries are required in a microservice architecture.

Asynchronous Message-Passing and Microservices

Asynchronous message-passing plays a significant role in keeping things loosely coupled in a microservice architecture. We used a message broker to deliver event notifications from our Shipment Management microservice to the Shipment Tracking microservice in an asynchronous manner. If two microservices are directly communicating via a message-queue channel, they are sharing a data space (the channel) and we have already talked, at length, about the evils of two microservices sharing a data space. Instead, what we can do is encapsulate message-passing behind an independent microservice that can provide message-passing *capability*, in a loosely coupled way, to all interested microservices.

The message-passing workflow we are most interested in, in the context of microservice architecture, is a simple publish/subscribe workflow. How do we express it as an HTTP API/microservice in a standard way? We recommend basing such a workflow on an existing standard, such as PubSubHubbub. Now to be fair, PubSubHubbub wasn't created for APIs or hypermedia APIs, it was created for RSS and Atom feeds in the blogging context. That said, we can adapt it relatively well to serve a hypermedia API-enabled workflow. To do so, we need to implement a flow similar to the one shown in below figure.

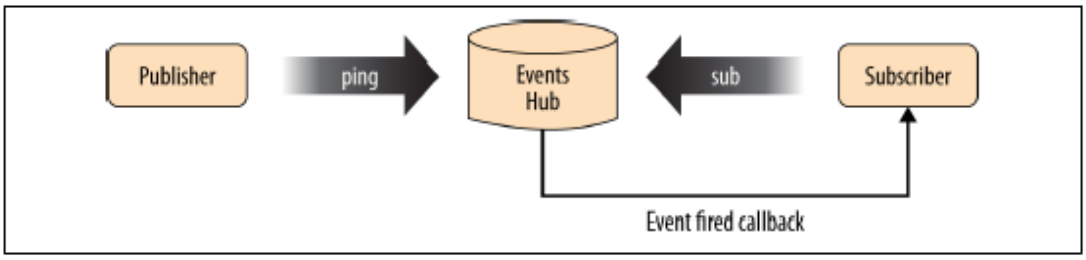

Figure 5-5. Asynchronous message-passing implemented with a PubSubHubbub-inspired flow

Dealing with Dependencies

Let's imagine that Shipping, Inc.'s currency rates microservice is being hammered by user queries and requests from other microservices. It would cost us much less if we hosted that microservice in a public cloud rather than on expensive servers of our corporate data center. But it doesn't seem possible to move the microservice to another host, if it stores data in the same SQL or NoSQL database system as all other microservices.

A strict requirement of full dependency embedding can be a significant problem, since for decades we have designed our architectures with centralized data storage, as shown in below figure.

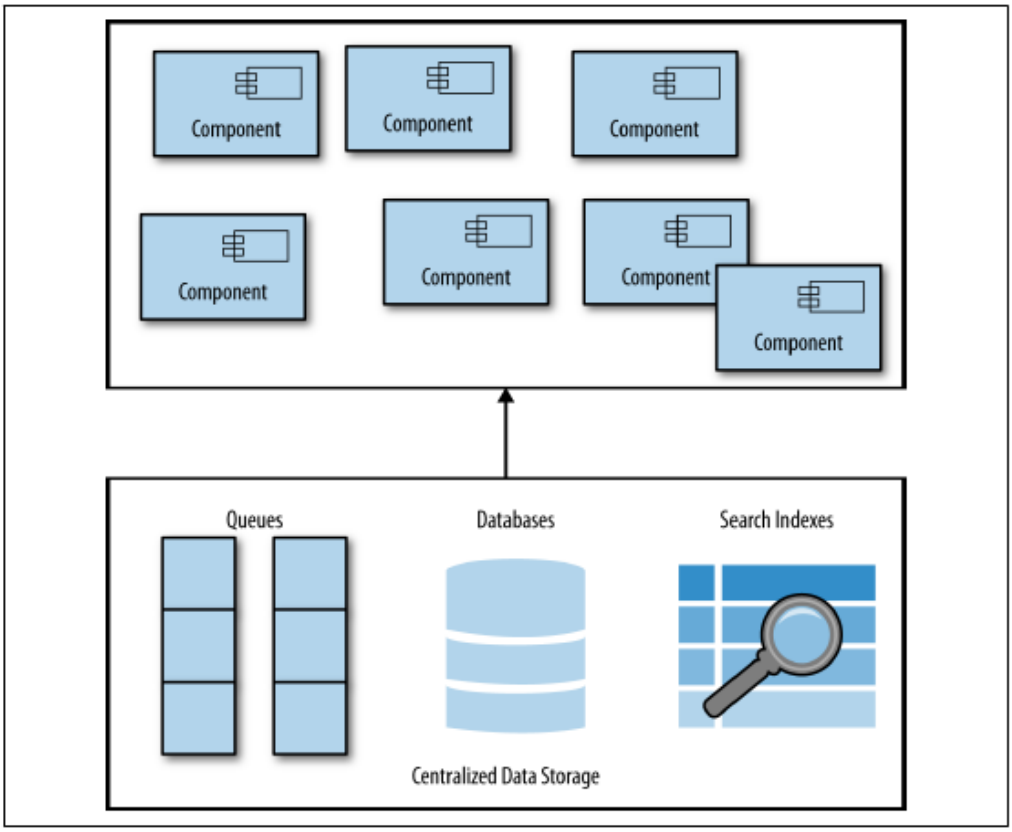

Figure 5-6. Components using a centralized pool of dependencies

Centralized data storage is operationally convenient: it allows dedicated, specialized teams (DBAs, sysadmins) to maintain and fine-tune these complex systems, obscuring the complexity from the developers.

In contrast, microservices favor *embedding of all their dependencies*, in order to achieve independent deployability. In such a scenario, every microservice manages and embeds its database, key-value store, search index, queue, etc. Then moving this microservice anywhere becomes trivial. This deployment would look like below figure.

The postulate of wholesale embedding of (data storage) dependencies looks beautiful on the surface, but in practice it is extremely wasteful for all but the simplest use cases. It is obvious that you will have a very hard time embedding entire Cassandra, Oracle, or ElasticSearch clusters in each and every microservice you develop.

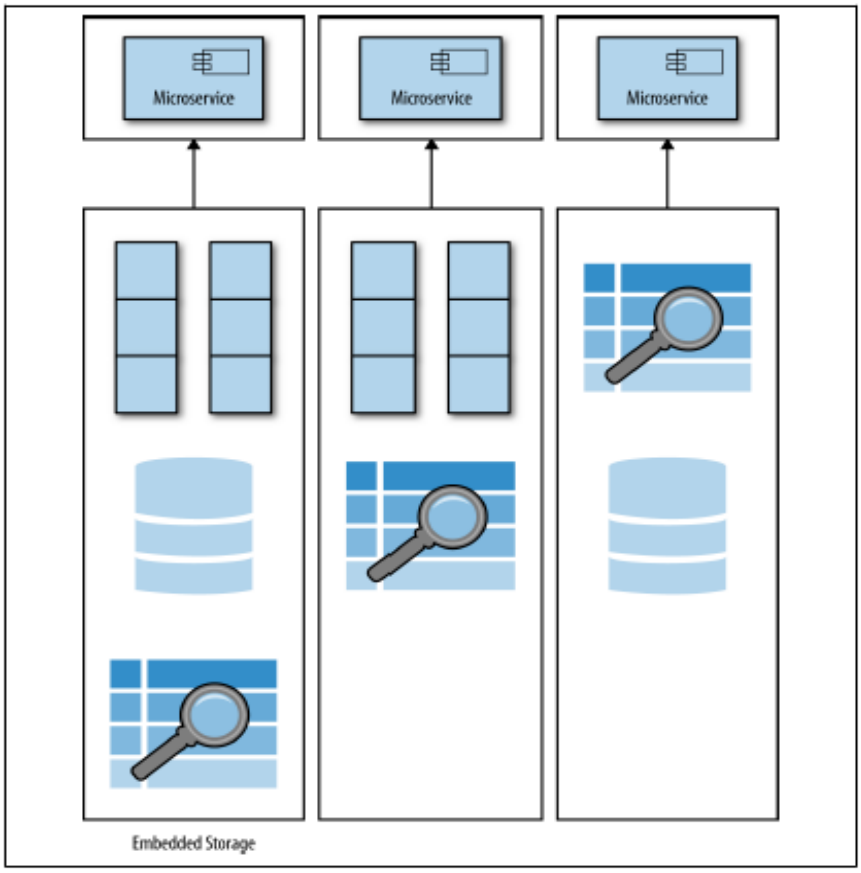

Figure 5-7. Components using fully embedded, isolated dependencies

Pragmatic Mobility

Figure 5-8 shows what a proper, sophisticated microservices deployment should look like in practice.

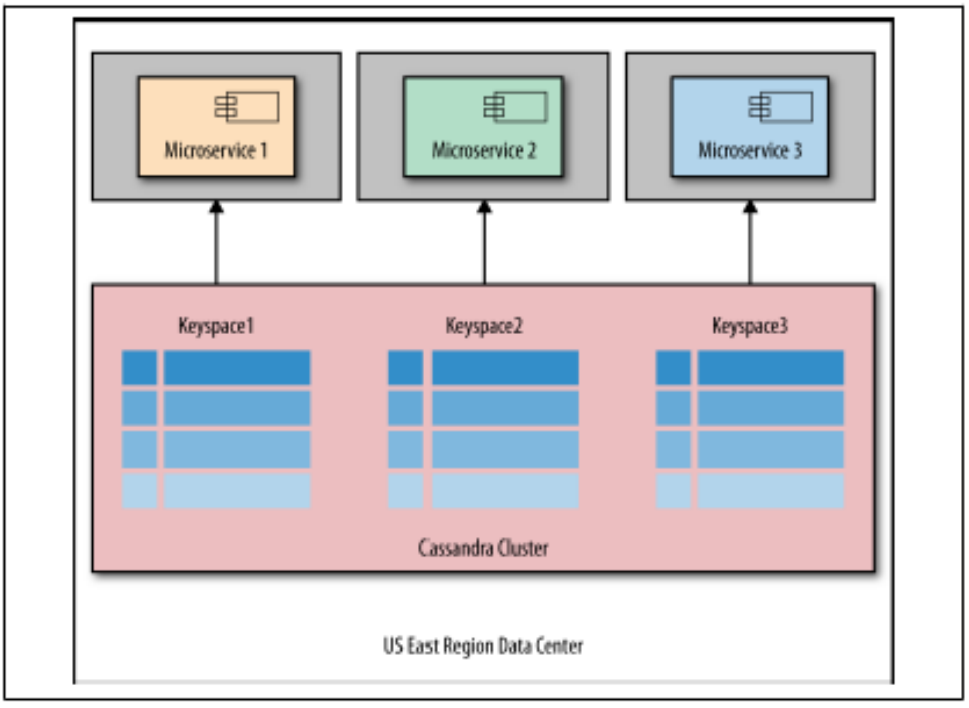

Figure 5-8. Pragmatic approach: Components using a centralized pool of dependencies, without sharing data spaces

If we decide to move Microservice 1 to another data center, it will expect that the new data center also has a functioning Cassandra cluster with a compatible version but it will find a way to move its slice of data and won't depend on the existence or state of any other microservice at the destination.

Microservices do not have to "travel" heavy and pack everything they may possibly require. In complicated cases it is OK to have some reasonable expectations about the destination environment, especially when it comes to complex data-storage capabilities.

The most important question we need to ask, when deciding on embedding dependencies versus "expecting" traits in an environment, is will our decision increase or decrease mobility? Our goal is to maximize deployment mobility of a microservice, which may mean different things in different contexts.

System Design and Operations

Independent Deployability

One of the core principles of the microservice architectural style is the principle of independent deployability—i.e., each microservice must be deployable completely independent of any other microservice. Independent deployability allows us to perform selective or on-demand scaling; if a part of the system (e.g., a single microservice) experiences high load we can re-deploy or move that microservice to an environment with more resources, without having to scale up hardware capacity for the entire, typically large, enterprise system. For many organizations, the operational ability of selective scaling can save large amounts of money and provide essential flexibility.

Scaling hardware resources on-premise can be extremely costly—we have to buy expensive hardware in *anticipation* of the usage rather than in response to actual usage. At the same time, the part of the application that gets hammered under load and needs scaling may not contain any sensitive client or financial data. It can be something as trivial as an API returning a list of US states or an API capable of converting various currency rates. The chief architect of Shipping, Inc. is confident that their security team will easily allow deployment of such safe microservices to a public/private cloud, where scaling of resources is significantly cheaper. The question is— could they deploy part of an application to a separate data center, a cloud-based one, in this case? The way most, typically monolithic, enterprise systems are architected, deploying selective parts of the application independently is either very hard or practically impossible. Microservices, in contrast, emphasize the requirement of independent deployability to the level of core principle, thus giving us much needed operational flexibility.

On top of operational cost saving and flexibility, another significant benefit of the independent deployability is an organizational one. Generally speaking, two different teams would be responsible for the development of separate microservices (e.g., Customer Management and Shipment Management). If the first team, which is responsible for the Customer Management microservice, needs to make a change and rerelease, but Customer Management cannot be released independent of the Shipment Management microservice, we now need to coordinate Customer Management's release with the team that owns Shipment Management. Such coordination can be costly and complicated, since the latter team may have completely different priorities from the team responsible for Customer Management. More often than not the necessity of such coordination will delay a release. Now imagine that instead of just a handful we potentially have hundreds of microservices maintained by dozens of teams. Release coordination overhead can be devastating for such organizations, leading to products that ship with significant delays or sometimes get obsolete by the time they can be shipped. Eliminating costly cross-team coordination challenges is indeed a significant motivation for microservice adopters.

More Servers, More Servers!

Let's assume we are developing a Java/JEE application. At first glance, something like a WAR or EAR file may seem like an appropriate unit of encapsulation and isolation. After all, that's what these packaging formats were designed for—to distribute a collection of executable code and related resources that together form an independent application, within the context of an application server.

In reality, lightweight packaging solutions, such as JAR, WAR, and EAR archives in Java don't provide sufficient modularity and the level of isolation required for microservices. WAR files and Gem files still share system resources like disk, memory, shared libraries, the operating system, etc. One of the core motivations of adopting a microservice architecture is to avoid the need for complex coordination and conflict resolution, thus packaging solutions that are incapable of avoiding such interdependencies are not suitable for microservices. We need a higher level of component isolation to guarantee independent deployability.

What if we deployed a microservice per physical server or per virtual machine? Well, that would certainly meet the high bar of isolation demanded by microservices, but what would be the financial cost of such a solution?

For companies that have been using microservice architecture for a number of years, it is not uncommon to develop and maintain hundreds of microservices. Typically, most companies run more than one environment (QA, stage, integration, etc.), which quickly multiplies the number of required servers.

Here comes the bad news: thousands of servers cost a lot. Even if we use VMs and not physical servers, even in the "cheapest" cloud-hosting environment the budget for a setup utilizing thousands of servers would be significantly high, probably higher than what most companies can afford or would like to spend. And then there's that important question of development environments. Most developers like to have a working, complete, if scaled down, model of the production environment on their workstations. How many VMs can we realistically launch on a single laptop or desktop computer? Maybe five or ten, at most? Definitely not hundreds or thousands.

So, what does this quick, on-a-napkin-style calculation of microservices hosting costs mean? Is a microservice architecture simply unrealistic and unattainable, from an operational perspective? It probably was, for most companies, some number of years ago. And that's why you see larger companies, such as Amazon and Netflix, being the pioneers of the architectural style—they were the few who could justify the costs. Things, however, have changed significantly in recent years.

The reason microservice architecture is financially and operationally feasible has a lot to do with containers.

The deployment unit universally used for releasing and shipping microservices is a container. If you have never used containers before, you can think of a container as of an extremely lightweight "virtual machine." The technology is very different from that of conventional VMs. It is based on a Linux kernel extension (LXC) that allows running many isolated Linux environments (containers) on a single Linux host sharing the operating system kernel, which means we can run hundreds of containers on a single server or VM and still achieve the environment isolation and autonomy that is on par with running independent servers, and is therefore entirely acceptable for our microservices needs.

Docker and Microservices

Docker is *the* container toolset most widely deployed in production today. At the beginning of 2016 most microservices deployments are practically unthinkable without utilizing Docker containers. But we shouldn't think of Docker or containers as tools designed just for the microservice architecture.

Containers in general, and Docker specifically, certainly exist outside microservice architecture. As a matter of fact, if we look at the current systems operations landscape we can see that the number of individuals and companies using containers many times exceeds those implementing microservice architecture. Docker in and of itself is significantly more common than the microservice architecture.

Containers were not created for microservices. They emerged as a powerful response to a practical need: technology teams needed a capable toolset for universal and predictable deployment of complex applications. Indeed, by packaging our application as a Docker container, which assumes prebundling all the required dependencies at the correct version numbers, we can enable others to reliably deploy it to any cloud or on-premise hosting facility, without worrying about target environment and compatibility. The only remaining deployment requirement is that the servers should be Docker-enabled—a pretty low bar, in most cases.

Linux containers use a layered filesystem architecture known as union mounting. This allows a great extensibility and reusability not found in conventional VM architectures. With containers, it is trivial to extend your image from the “base image" If the base image updates, your container will inherit the changes at the next rebuild. Such a layered, inheritable build process promotes a collaborative development, multiplying the efforts of many teams. Centralized registries, discovery services, and community- oriented platforms such as Docker Hub and GitHub further facilitate quick adoption and education in the space.

As a matter of fact, we could easily turn the tables and claim that it is Docker that will be driving the adoption of microservices instead of vice versa. One of the reasons for this claim is that Docker puts significant emphasis on the “Unix philosophy” of shipping containers, i.e., “do one thing, and do it well” Indeed, this core principle is prominently outlined in the Docker documentation itself:

> Run only one process per container. In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers.

It is clear that with such principles at its core Docker philosophy is much closer to the microservice architecture than a conventional, large monolithic architecture. When you are shooting for “doing one thing” it makes little sense to containerize your entire, huge, enterprise application as a single Docker container. Most certainly you would want to first modularize the application into loosely coupled components that communicate via standard network protocols, which, in essence, is what the microservice architecture delivers. As such, if you start with a goal of containerizing your large and complex application you will likely need a certain level of microservice design in your complex application.

The way we like to look at it, Docker containers and microservice architecture are two ends of the road that lead to the same ultimate goal of continuous delivery and operational efficiency. You may start at either end, as long as the desired goals are achieved.

The Role of Service Discovery

If you are using Docker containers to package and deploy your microservices, you can use a simple Docker Compose configuration to orchestrate multiple microservices (and their containers) into a coherent application. As long as you are on a single host (server) this configuration will allow multiple microservices to “discover” and communicate with each other. This approach is commonly used in local development and for quick prototyping.

But in production environments, things can get significantly more complicated. Due to reliability and redundancy

needs, it is very unlikely that you will be using just one Docker host in production. Instead, you will probably deploy at least three or more Docker hosts, with a number of containers on each one of them.

Furthermore, if your services get significantly different levels of load, you may decide to not deploy all services on all hosts but end up deploying high-load services on a select number of hosts (let's say ten of them), while low-load services may only be deployed on three servers, and not necessarily the same ones. Additionally, there may be security- and business-related reasons that may cause you to deploy some services on certain hosts and other services on different ones.

In general, how you distribute your services across your available hosts will depend on your business and technical needs and very likely may change over time. Hosts are just servers, they are not guaranteed to last forever.

Figure 6-1 shows what the nonuniform distribution of your services may look like at some point in time if you have four hosts with four containers.

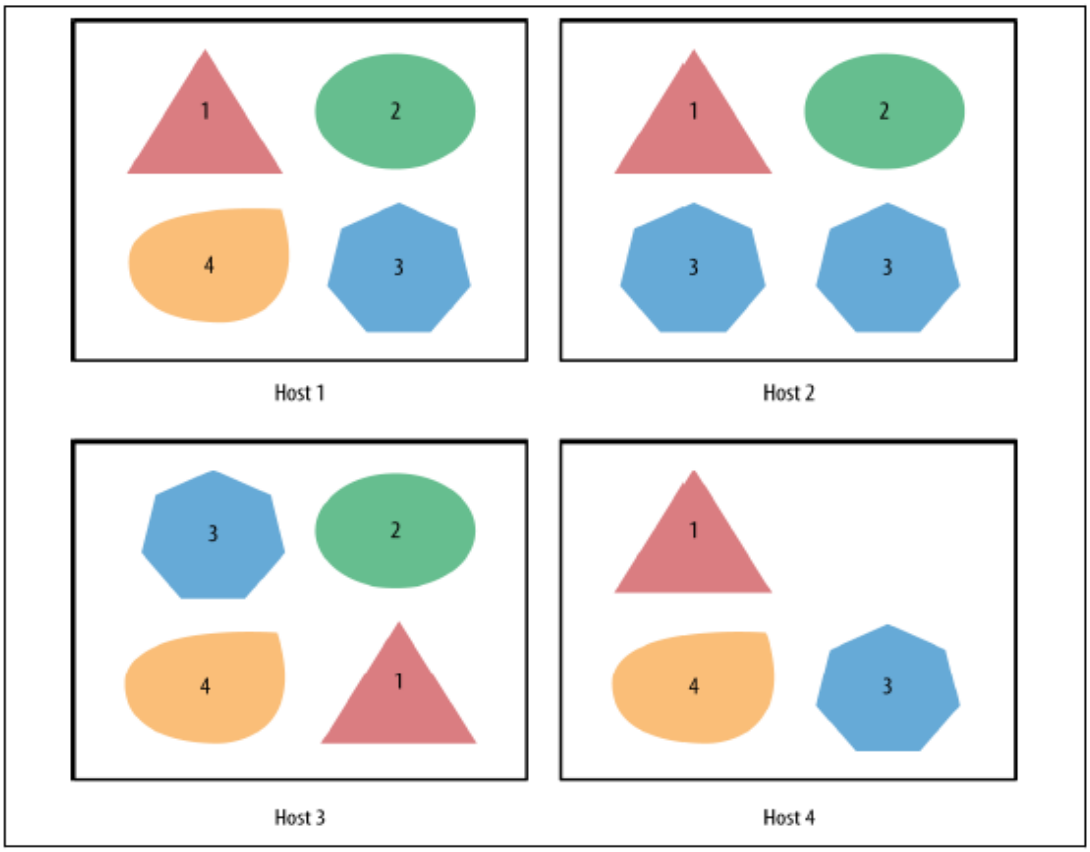

Figure 6-1. Microservice deployment topology with nonuniform service distribution

Each instance of the microservice container in Figure 6-1 is depicted with a different number, shape, and color. In this example, we have Microservice 1 deployed on all four hosts, but Microservice 2 is only on hosts 1-3. Keep in mind that the deployment topology may change at any time, based on load, business rules, which host is available, and whether an instance of your microservice suddenly crashes or not.

Note that since typically many services are deployed on the same host, we cannot address a microservice by just an IP address. There are usually too many microservices, and the instances of those can go up and down at any time. If we allocated an IP per microservice, the IP address allocation + assignment would become too complicated. Instead, we allocate an IP per host (server) and the microservice is fully addressed with a combination of:

1. IP address (of the host)
2. Port number(s) the service is available at on the host

If we assumed that a specific microservice always launches on a specific port of a host, we would require a high level of cross-team coordination to ensure that multiple teams don't accidentally claim the same port. But one of the main motivations of using a microservice architecture is eliminating the need for costly crossteam coordination. Such coordination is untenable, in general. It is also unnecessary since there are better ways to achieve the same goal.

This is where service discovery enters the microservices scene. We need some system that will keep an eye on all services at all times and keep track of which service is deployed on which IP/port combination at any given time, so that the clients of microservices can be seamlessly routed accordingly.

The Need for an API Gateway

We will look into each one of the API gateway features and clarify their role in the overall architecture of the operations layer for microservices.

Security

Microservice architecture is an architecture with a significantly high degree of freedom. Or in other words, there are a lot of more moving parts than in a monolithic application. As we mentioned earlier, in mature microservices organizations where the architecture is implemented for complex enterprise applications, it is common to have hundreds of microservices deployed. Things can go horribly wrong security- wise when there are many moving parts. We certainly need some law and order to keep everything in control and safe. Which is why, in virtually all microservice implementations, we see API endpoints provided by various microservices secured using a capable API gateway.

APIs provided by microservices may call each other, may be called by "frontend," i.e., public-facing APIs, or they may be directly called by API clients such as mobile applications, web applications, and partner systems. Depending on the microservice itself, the business needs of the organization, and the industry, market, or application context—all scenarios are fair game. To make sure we never compromise the security of the overall system, the widely recommended approach is to secure invocation of "public-facing" API endpoints of the microservices-enabled system using a capable API gateway.

In reality the distinction between "public" and "private" APIs often ends up being arbitrary. How certain are we that the API we think is "only internal" will never be required by any outside system? As soon as we try to use an API over the public Web, from our own web application or from a mobile application, as far as security is concerned, that endpoint is "public" and needs to be secured.

Sometimes, certain microservices are deemed "internal" and excluded from the security provided by an API Gateway, as we assume that they can never be reached by external clients. This is dangerous since the assumption may, over time, become invalid. It's better to always secure any API/microservice access with an API gateway. In most cases the negligible overhead of introducing an API gateway in between service calls is well worth the benefits.

Transformation and Orchestration

We know that microservices are typically designed to provide a single capability. They are the Web's version of embracing the Unix philosophy of "do one thing, and do it well." However, as any Unix developer will tell you, the single responsibility approach only works because Unix facilitates advanced *orchestration* of its highly specialized utilities, through universal piping of inputs and outputs. Using pipes, you can easily combine and chain Unix utilities to solve nontrivial problems involving sophisticated process workflows. Basically, to make microservices useful, we need an orchestration framework like Unix piping, but one geared to web APIs.

Microservices, due to their narrow specialization and typically small size, are very useful deployment units for the teams producing them. That said, they may or may not be as convenient for consumption, depending on the client. The Web is a distributed system. Due to its distributed nature, on the Web, so-called "chatty" interfaces are shunned. Those are interfaces where you need to make many calls to get the data required for a single task. This distaste for chatty interfaces is especially pronounced among mobile developers, since they often have to deal with unreliable, intermittent, and slow connections.

Let's imagine that after successful completion of the APIs required for the mobile application, the technical team behind Shipping, Inc.'s microservice architecture decided to embark on a new journey of developing an "intelligent" inventory management system. The purpose of the new system is to analyze properly anonymized data about millions of shipments passing through Shipping, Inc., combine this insight with all of the metadata that is available on the goods being shipped, determine behavioral patterns of the consumers, and utilizing human + machine algorithms design a "recommendation engine" capable of suggesting optimal inventory levels to Shipping, Inc.'s "platinum" customers. If everything works, those suggestions will be able to help customers achieve unparalleled efficiency in managing product stock, addressing one of the main concerns of any online retailer.

If the team is building this system using a microservice architecture, they could end up creating two microservices for the main functionality:

1. Recommendations microservice, which takes user information in, and responds with the list containing the recommendations—i.e., suggested stock levels for various products that this customer typically ships.
2. Product Metadata microservice, which takes in an ID of a product type and retrieves all kinds of useful metadata about it.

Such separation of concerns, into specialized microservices, makes complete sense from the perspective of the API publisher, or as we may call them, the server-side team. However, for the team that is actually implementing the end-user interface, calling the preceding microservices is nothing but a headache. More likely than not, the mobile team is working on a user screen where they are trying to display several upcoming suggestions. Let's say the page size is 20, so 20 suggestions at a time. With the current, verbatim design of the microservices, the user-interface team will have to make 21 HTTP calls: one to retrieve the recommendations list and then one for each recommendation to retrieve the details, such as product name, dimensions, size, price, etc.

At this point, the user-interface team is not happy. They wanted a single list; but instead are forced to make multiple calls (the infamous "N+1 queries" problem, resurfaced in APIs). Additionally, the calls to the Product Metadata microservice return too much information (large payload problem), which is an issue for, say, mobile devices on slow connections. And the end result is that the rendering of the all-important mobile screen is slow and sluggish, leading to poor user experience.

Routing

We already mentioned that in order to properly discover microservices we need to use a service discovery system. However, directly providing tuples of the IP/port combinations to route an API client is not an adequate solution. A proper solution needs to abstract implementation details from the client. An API client still expects to retrieve an API at a specific URI, regardless of whether there's a microservice architecture behind it and independent of how many servers, Docker containers, or anything else is serving the request.

Some of the service discovery solutions (e.g., Consul, and etcd using SkyDNS) provide a DNS-based interface to discovery. This can be very useful for debugging, but still falls short of production needs because normal DNS queries only look up domain/IP mapping, whereas for microservices we need domain mapping with an IP +port combination. We should do is let an API gateway hide the complexities of routing to a microservice from the client apps. An API gateway can interface with either HTTP or DNS interfaces of a service discovery system and route an API client to the correct service when an external URI associated with the microservice is requested. You can also use a load balancer or smart-reverse proxy to achieve the same goal, but since we already use API gateways to secure routes to microservices, it makes a lot of sense for the routing requirement to also be implemented on the gateway.

Monitoring and Alerting

As we have already known, while microservice architecture delivers significant benefits, it is also a system with a lot of more moving parts than the alternative—monolith. As such, when implementing a microservice architecture, it becomes very important to have extensive system-wide monitoring and to avoid cascading failures.

The same tools that we mentioned for service discovery can also provide powerful monitoring and failover capabilities. Let's take Consul as an example. Not only does Consul know how many active containers exist for a specific service, marking a service broken if that number is zero, but Consul also allows us to deploy customized health-check monitors for any service. This can be very useful. Indeed, just because a container instance for a microservice is up and running doesn't always mean the microservice itself is healthy. We may want to additionally check that the microservice is responding on a specific port or a specific URL, possibly even checking that the health ping returns predetermined response data.

In addition to the "pull" workflow in which Consul agents query a service, we can also configure "push"-oriented health checks, where the microservice itself is responsible for periodically checking in, i.e., push predetermined payload to Consul. If Consul doesn't receive such a "check-in," the instance of the service will be marked "broken." This alternative workflow is very valuable for scheduled services that must run on predetermined schedules. It is often hard to verify that scheduled jobs do run as expected, but the "push"-based health-check workflow can give us exactly what we need.

Adopting Microservices in Practice

Solution Architecture Guidance

Solution architecture is distinct from individual service design elements because it represents a macro view of our solution. Here are some issues you may encounter when working at this macro-level view of the system.

- **How many bug fixes/features should be included in a single release?**

Since releases are expected to happen frequently, each release will likely be small. You probably can't box

up 50 changes to a single service component in a week. We hear most organizations have a practice of limiting the number of changes. Netflix, for example, tells teams to make only one significant change per release. For example, if your team needs to refactor some of the internal code *and* start using a new data store module, that would be two releases.

The biggest reason for limiting the number of changes in a release is to reduce uncertainty. If you release a component that contains multiple changes, the uncertainty is increased by the number of interactions that occur between those changes. The mathematical discipline of graph theory provides a simple formula to calculate those interactions: n(n-l)/2. Based on this, if you release a component that contains 5 changes and it causes problems in production, you know that there are 10 possible ways in which these 5 changes could interact to cause a problem. But if you release a component with l5 changes there is a potential for over l00 different ways in which those changes can interact to cause problems—and that's just internal problems.

Limit the number of changes in each release to increase the safety of each release.

- **When do I know our microservice transformation is done?**

Technically, creating and maintaining a vital information system is never "done." And that is also true for one built in the microservices way. In our experience some architects and developers spend a lot of time trying to identify the ideal solution or implementation model for their system design. It rarely works. In fact, one of the advantages of microservices is that change over time is not as costly or dangerous as it might be in tightly coupled large-scope release models.

Trying to perfect "the system" is an impossible task since it will always be a moving target. Often arriving at some "final state" marks the start of accumulating "technical debt"—that status where the system is outdated and difficult to change. It helps to remember that everything you build today will likely be obsolete within a few years anyway.

Since doing things in the microservices way means lots of small releases over time, you'll always be changing/improving something. This means you get lots of "done" moments along the way and, in keeping with the theme of microservices, are able to effect change over time "at scale."

Organizational Guidance

From a microservice system perspective, organizational design includes the structure, direction of authority, granularity, and composition of teams. A good microservice system designer understands the implications of changing these organizational properties and knows that good service design is a byproduct of good organizational design.

How do I know if my organization is ready for microservices?

You can start by assessing your organization's structure and associated culture. The majority of microservice architecture pioneers began their quest for faster software delivery by optimizing organizational design before addressing the software architecture. Given this progression, these organizations landed on microservice architecture as a style that aligned with their small, business-aligned teams, thus revealing the wisdom of Conway's decades-old assertion.

However, many organizations now evaluating microservice architecture are not following the same path. In those cases, it is crucial to look at the organizational structure.

- How are responsibilities divided between teams?
- Are they aligned to business domains, or technology skillsets?

- At what level of the organization are development and operations divided?
- How big are the teams?
- What skills do they have?
- How dynamic is the communication and interaction between the teams who need to be involved in the delivery lifecycle?
- How is power distributed between the teams?
- Is it centralized at a high level, or decentralized among the delivery teams?

Answering these questions will help you understand what impacts these organizational factors will have on your adoption efforts and resulting successes.

The ideal organization for microservices has small, empowered teams responsible for services that align with specific business domains. These teams feature all of the roles necessary to deliver these services, such as product owners, architects, developers, quality engineers, and operational engineers. The teams also need the right skills, such as API design and development, and knowledge of distributed applications. Organizations that mismatch any of these characteristics will pay a toll when attempting to apply microservice architecture. Lastly, if the team doesn't have the right skills to build API-fronted services using distributed concepts, costs could go up to cover training and/or contract hiring, or the solution could be dragged away from the microservices approach as existing resources retreat to their technological comfort zones.

Culture Guidance

Your organization's culture is important because it shapes all of the atomic decisions that people within the system will make. This large scope of influence is what makes it such a powerful tool in your system design endeavor.

How do I introduce change?

If you aren't working in a *greenfield* environment, chances are you'll have inherited an existing organizational design as well. Making changes to a working organization is nontrivial and carries a much greater risk than toying with a solution architecture. After all, if we make a mistake when refactoring our software we can always undo our changes, but when we make a mistake when redesigning the reporting structure in an organization the damage is not so easily undone.

In order to apply a refactoring strategy to the organizational design, you'll need to:

1. Devise a way to test changes
2. Identify problem areas in your organizational design
3. Identify safe transformations (changes that don't change existing behavior)

Refactoring the organization won't help you do something you don't already know how to do. Your goal should be to do the same things, but improve the design of your organization so you can do them better.

When you refactor an application you can measure and observe the performance of the application; you can audit the source code, and you can comb through logs and determine where most of the problems occur. But when dealing with the processes and people that make up an organization things are a bit less black and white. To successfully identify where the refactoring opportunities are within the organization, you'll need to find some way to model the existing system in order to analyze and measure its performance.

For the microservice system we are especially interested in identifying opportunities to improve the

efficiency of change. Gaining a total understanding of how your organization works may be too large of an initial investment to undertake, so in practice you may need to focus only on the changes that occur the most often for the components that are the most volatile.

In particular, you should be looking for the bottlenecks that cause change to be expensive. Which processes result in a queue or backlog? Are there particular centralized functions such as audits, code reviews, and gating procedures that cause teams to have to wait? Are there any parts of your process flow that make it difficult for multiple changes to be introduced at the same time due to resource availability or a need for serialized process execution? Finding these bottlenecks will help you identify good candidates for process and organizational refactoring as they should yield a large benefit to the changeability and speed of release for the system.

Can I do microservices in a project-centric culture?

A hallmark of a microservices organization is that the teams that implement a feature, application, or service continue to support, improve, and work on the code for its lifetime. This product-centric perspective instills a sense of ownership of the component and reinforces the idea that deployed components will constantly be updated and replaced.

Typical project-centric cultures operate differently. Teams are formed to address a particular problem (e.g., create a new component, add a feature, etc.) and disbanded when that problem is solved. Often a good deal of knowledge about both the problem and the solution gets lost when the team disbands. And, if there is a need to readdress the same problem, or make additional changes to the same component, it may be difficult to re-create the team or recover the lost knowledge. These challenges usually mean changes happen less often and are more likely to result in bugs or partial solutions.

In truth, it is quite difficult to adopt the microservice style if you need to operate in this type of culture. If changeability and speed of release are important properties for your system, the long-term goal should be to transition to a style of building that encourages team-based ownership of components.

Can I do microservices with outsourced workers?

A particular challenge for large companies trying to incorporate the ideal microservices system is the trend toward outsourcing technology services. The act of hiring an outside company to perform development and operations activities using workers who are external to the organization seems at odds with the culture and organizational principles we've described in this book. But with the right outsourcing structure, a microservice system may lend itself well to being developed by an external organization.

By embracing a decentralized way of working and standardizing on the output and processes of service teams (containers and APIs), the outsourced development team can be given enough autonomy to build a service that meets the capability requirements of the owning organization. But this is only possible if the outsourced team conforms to the principles that exemplify the microservices way: the teams should be the *right* size, built to last for the perpetuity of the life of the service, and composed of workers who are skilled, experienced, and capable enough to make good design and implementation decisions autonomously.

In addition to team composition, the microservice designer should acknowledge that a cross-pollination of cultures occurs whenever outsourcing is conducted. The implication is that a desired organizational culture cannot simply be adopted by the outsourced team, nor can the buying organization avoid having their culture changed by the intermingling of work. This means that culture becomes an important element in deciding

which companies or people should be chosen to support the outsourcing model.

Ultimately, the selection process for a microservices outsourcing model cannot be optimized purely for low-cost work. You will need to carefully select a partner who is amenable to the cultural traits you are looking for and possesses aspects of culture you'd like to incorporate into your own system. The deal must also be structured to incentivize the team dynamic that works best for building applications the microservices way—teams should be dedicated to services, workers should be capable of working autonomously, and speed of high-quality delivery should be the primary metric for success.

Tools and Process Guidance

The system behavior is also a result of the processes and tools that workers in the system use to do their job. In microservices systems, this usually includes tooling and processes related to software development, code deployment, maintenance, and product management.

What kinds of tools and technology are required for microservices?

The ideal technological environment for microservices features cloud infrastructure, which facilitates rapid provisioning and automated deployment. The use of containers is particularly useful to enable portability and heterogeneity. Middleware for data storage, integration, security, and operations should be web API-friendly in order to facilitate automation and discovery, and should also be amenable to dynamic, decentralized distribution. The ideal programming languages for microservices are API friendly as well, and should be functional while also matching the skillsets of your organization. It is particularly useful to provide tools for developers that simplify their tasks yet incorporate constraints that encourage good operational behavior of their resulting code.

Straying from these technological traits can lead to adoption issues. Lack of cloud infrastructure will lead to deployment delays and inflexible scaling. Lack of containers—or reliance on older virtualization or app servers—could increase the cost of resource utilization and lead to quality issues resulting from inconsistencies across environments. Middleware that assumes strict centralized control will break the decentralized organizational model and challenge the provisioning of ephemeral environments. If used in a decentralized model, this specialized middleware could also lead to skill challenges in the organization if every team is required to cultivate expertise. Centralized or segregated data breaks the organizational model as well. It also slows down delivery and impedes evolvability. Lack of developer tooling consistency could lead to duplicate work and lack of visibility or resiliency in the overall system. Finally, a large dependency on legacy applications could limit the ability to make changes.

What kinds of practices and processes will I need to support microservices?

The ideal software development lifecycle for microservices is based on a product mentality using Agile principles, which includes continuous integration and continuous delivery and features a high degree of automation in testing, deployment, and operations. Attempting to apply microservice architecture in a differing environment can subtract from its potential value. A Waterfall approach can lead to tight coupling of services, making it difficult to manage the different change rates of those services and inhibiting their evolution. Project-focused delivery assumes static requirements and heavyweight change control, both impediments to fast software delivery. Being unable to deploy frequently will lead to a "big bang" release mentality and bring with it undue ceremony. If change frequency is increased in an environment that has a legacy of change intolerance, many of those overweight processes can stick around, slowing down delivery,

and introducing procedural fatigue as a new risk. Lack of automation in the deployment lifecycle will have a negative compound effect on speed to market, and lack of automation in operations will make it harder to deal with the operational complexity of a distributed environment.

How do I govern a microservice system?

Aside from regulatory issues (e.g., certification, audits, etc.) there are typically three ways in which you can address security and governance requirements in a microservice system: centralized, contextual, and decentralized.

Centralized controls

At the component level, there really isn't anything special about securing a microservice system. If you know how to secure an operating system, secure an API, or secure an application you can apply all of the same mechanisms to a microservice system. But when security mechanisms are introduced in the manner that most experts are used to implementing them, you can inadvertently upset the system optimization goals that you've worked hard to design into the architecture.

This is because security, controls, and governance policies are often implemented in a centralized fashion. For example, if we have a need to authenticate, authorize, and audit messages before they are processed, the most common architecture pattern is to implement some form of central security enforcement component within the architecture. Implementing a single, scalable component that can manage a complex and expensive function like access control makes a lot of sense. Assigning a separate team to manage and implement such a service also makes a lot of sense. Unfortunately, services like access control are likely to be used by every service in the infrastructure, which results in a common component that all of other services will grow dependent on. In other words, a bottleneck can develop.

A centralized security component risks putting our system into a state of *mechanical organization* or centralized control. In the early days of a microservice architecture it will be easy to set up the correct access and routing rules for a handful of services, but as more services are introduced and as those services change, the demand to modify the access control component is likely to outgrow the access control team's capacity to roll out changes in time.

Decentralized controls

The implication here is that the individual microservice teams will need to manage an infrastructure that includes security mechanisms that are bounded to the service itself. The organization may standardize on the particular components and libraries that are to be used in every microservice, but it will be the teams themselves that are responsible for implementing security components and configuring them accordingly. It naturally follows that someone on the team must also take on the role of becoming the security expert for the service.

Contextual control

A third approach that an organization can take is to define subsystems within the microservice architecture. Each subsystem may contain multiple services and their services within the subsystem are able to share common resources such as access control. Again, the organization may mandate the nature and requirement for these security components to be in place, but it is up to a subsystem service team to own and manage the configuration for the security component.

Security will always be an important design consideration for your microservice system. Even the absence of security is an implicit trade-off. While decisions about *how* and *what* to secure will be dependent on the

risk profile of your organization and nature of the application you are building, the decision about *who* will manage the security implementation and *where* it will be implemented will have a big impact on your ability to optimize for the system behavior that you want.

Services Guidance

In a microservice system, the services form the atomic building blocks from which the entire organism is built. The following are some additional questions and issues we've identified when implementing well-designed microservices and APIs.

Should all microservices be coded in the same programming language?

The short answer is "no" The internal language of the component is not as important as the external interface—the API—of that component. As long as two components can use the same network protocols to exchange messages in an agreed-upon format using shared terms, the programming language used to accomplish all this is not important.

At the same time, many companies we talked to constrained the number of languages supported in the organization in order to simplify support and training. While a polyglot environment has advantages, too many languages results in added nonessential complexity system developers and maintainers need to deal with.

What do I do about orphaned components?

Over the life of a microservice implementation teams will come and go, and sometimes a team might disband and this can result in an "orphaned" microservice. It's not a good idea to just let a service run along without someone to care for it. As Martin Fowler points out in "Products not Projects", "ownership" is an important organizational aspect of microservices.

When a team is about to disband, that team needs to designate a new "owner" of the microservice component. This might be one of the existing team members ("OK, I'll take responsibility for it"). It might be some other team that is willing to take care of it. Or it might be someone who has taken on the special role of caring for "orphaned" services. But someone needs to be designated as the "owner."

It's not safe to allow orphaned services to run in your infrastructure.