Microservices
Unit 2
Chapter 1: Service Design:

Q1) Microservice Boundaries?

- So just how micro should a microservice be?
- "What is the optimal size of a microservice?"
- **1. Microservice Boundaries and Domain-Driven Design (DDD):**
- 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.
- 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?
- The overall model of a large system is actually comprised of many smaller models that are intermingled with each other.
- **2. Bounded Context:**
- Teams need to be very careful when combining contextual models to form a larger software system.
- 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.
- Bounded contexts represent autonomous business domains 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.
- **3. Smaller Is Better:**
- Principle of size reduction: 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."
- Basically, 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).
- **4. Ubiquitous Language:**
- Bounded context should be as big as it needs to be in order to fully express its complete ubiquitous language.
- 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).

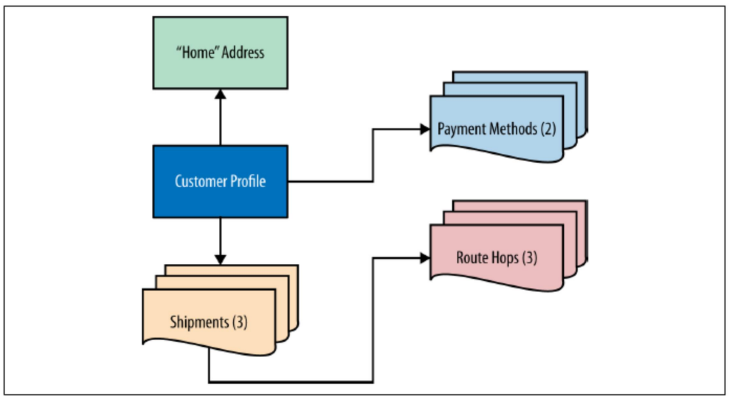
Q2) API design for Microservices?

- Microservice components only become valuable when they can communicate with other components in the system.
- They each have an interface or API.
- Just as we need to achieve a high level of separation, independence, and modularity of our code we need to make sure that our APIs, the component interfaces, are also loosely coupled.
- We see two practices in crafting APIs for microservices:
- **1. Message-oriented**
- 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.
- 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.
- **2.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.
- You may also know this style as APIs with Hypermedia As The Engine Of Application State (HATEOAS APIs).
- 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.
- 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.

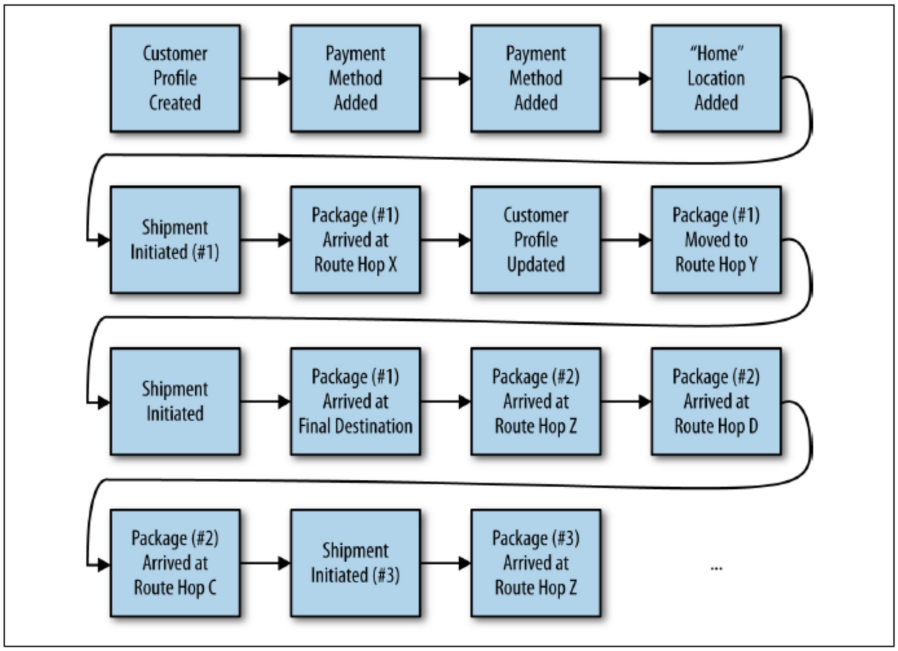
Q3)Data and Microservices?

- 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.
- But data-centric design is not a good way to implement distributed systems—especially systems that rely on independently deployable microservices.
- It turns out that capabilities-centric design is more suitable for microservices than a more traditional, data-centric design.
- **1. Example: 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.
- Company is very tech-savvy, 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.
- Accounting and sales subsystems (microservices) need access to daily currency exchange rates to perform their operations.
- A datacentric 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.
- 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.
- 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.
- Thinking in terms of capabilities rather than data is a very powerful technique for API design, in general.
- *techniques we can use to avoid data-sharing in complex use cases: event sourcing, CQRS—command query responsibility segregation
- **2. Event Sourcing:**
- 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.
- 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.
- Another crucial property of event sourcing: much like in life, we can never "go back" in time and "change" the past, we can only do something in the present to compensate for the mistakes of the past.

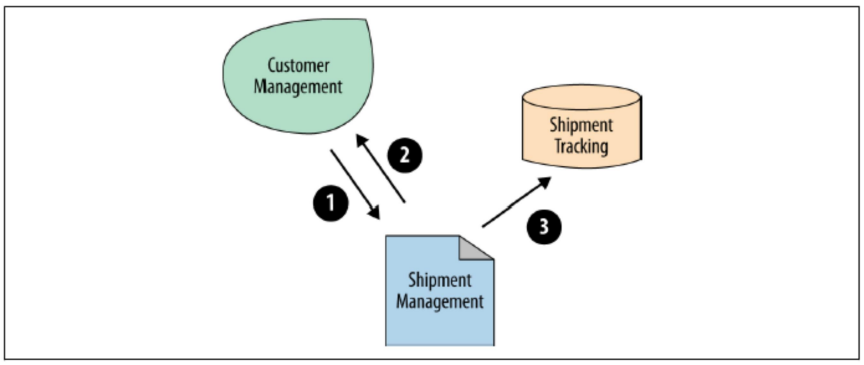
- In event sourcing, data is immutable—we always issue a new command/event to compensate rather than update a state of an entity, as we would do in a CRUD style.



- *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*
- **3. System Model for Shipping, Inc.**



- *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.
- **4.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.
- that data-altering operations should be in different methods, separated from methods performing read-only operations.
- CQRS takes this concept a large step further, instructing us to use entirely different models for updates versus queries.
- With CQRS, the Shipment Management microservice can "own" and encapsulate any updates related to package delivery, just notifying other contexts about events occurring.
- By subscribing to notifications of these events, a reporting service such as Shipment Tracking can build completely independent, query-optimized model(s) that don't need to be shared with any other service.
- Shipping Management doesn't even need to know about the existence of the Tracking microservice.
- During runtime the Tracking microservice only queries its own index.
- Furthermore, the Tracking microservice can include event and command data from other microservices using the same flow, keeping its independence and loose coupling.
- The big win with using event sourcing and CQRS is that they allow us to design very granular, loosely coupled components.

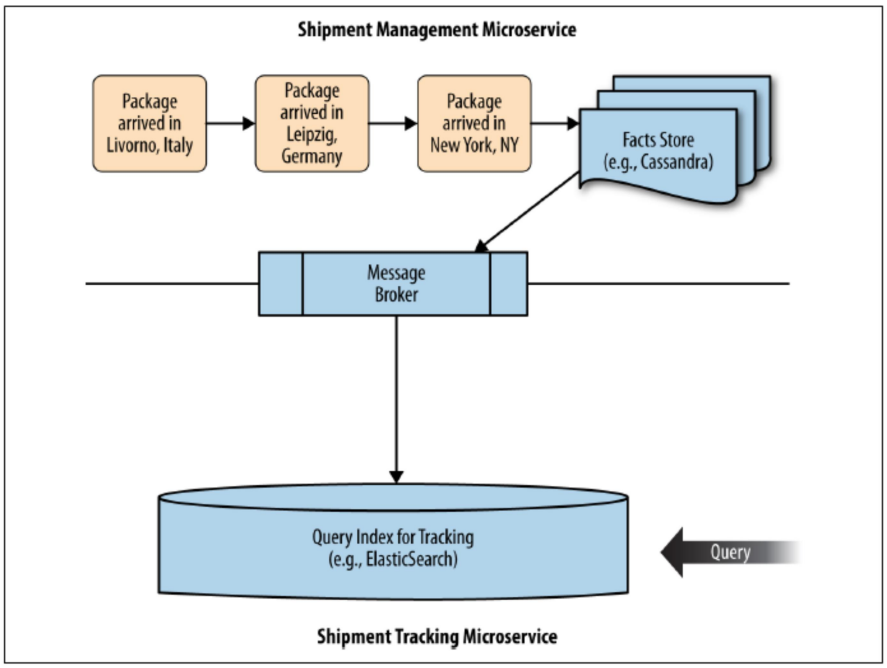


Figure 5-4. Data flow in command-query responsibility segregation (CQRS)-based model for Shipping, Inc.

-

Q4) Distributed Transactions and Sagas?

- 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.
- We call such processes "transactions."

- For distributed workflows and share-nothing environments (and microservice architecture is both of those), 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 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.
- 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.

Q5) Asynchronous Message-Passing and Microservices?

- Asynchronous message-passing plays a significant role in keeping things loosely coupled in a microservice architecture.
- We can 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.

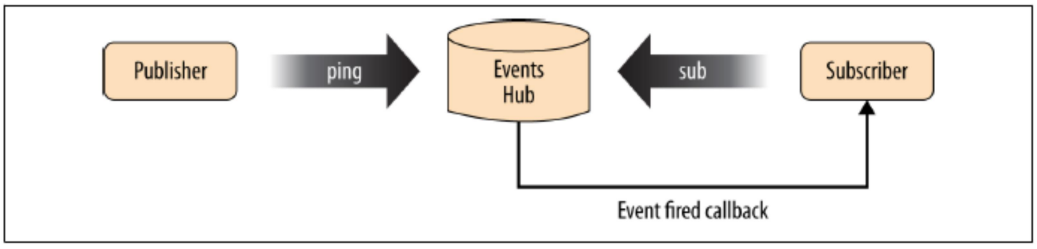


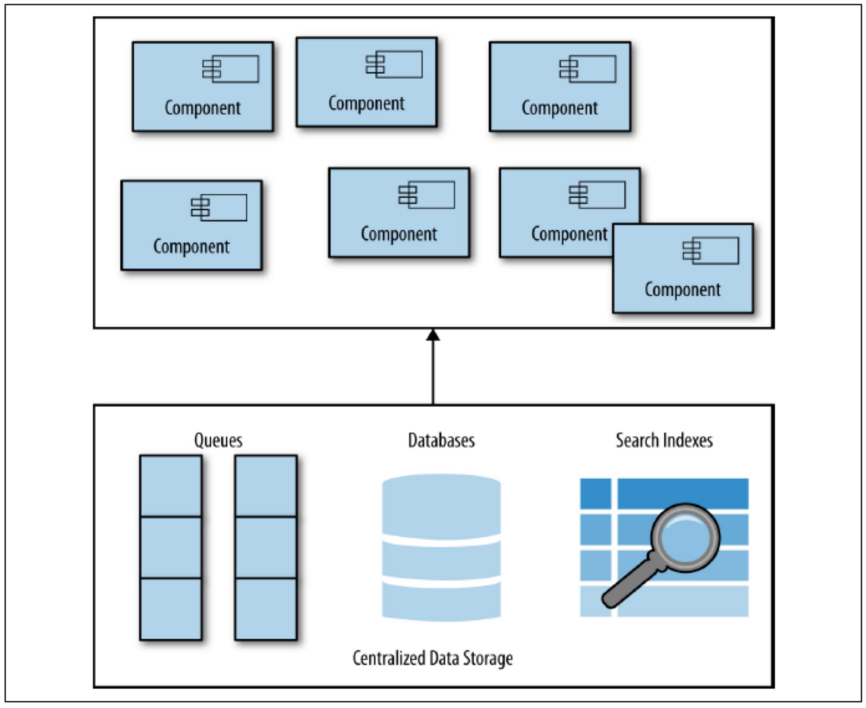
Figure 5-5. Asynchronous message-passing implemented with a PubSubHubbub-inspired flow

-
- We also need to standardize some hypermedia affordances:
- rel="hub"
- Refers to a hub that enables registration for notification of updates to the context.
- rel="pingback"
- Gives the address of the pingback resource for the link context.
- rel="sub"
- When included in a resource representation of an event, the "sub" (subscription) link relation may identify a target resource that represents the ability to subscribe to the pub/sub event-type resource in the link context.
- rel="unsub"

- When included in a resource representation of an event, the "unsub" (subscription cancellation) link relation may identify a target resource that represents the ability to unsubscribe from the pub/sub event-type resource in the link context.
- rel="event"
- Resource representation of a subscribable events.
- rel="events"
- Link to a collection resource representing a list of subscribable events.

Q6) Dealing with Dependencies?

- We cannot have any microservice share even the installation of a data storage system.
- Some may argue that a microservice needs to "embed" every single dependency it may require, so that the microservice can be deployed wherever and whenever, without any coordination with the rest of the system.



- *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.
- In reality, a microservice doesn't have to carry along every single dependency (such as a data storage system) in order to be mobile and freely move across the data centers.
- The trick to microservice mobility is not packing everything but instead ensuring that the deployment destination provides heavy assets, such as database clusters, in a usable and auto-discoverable form at every destination where a microservice may be deployed.
- Microservices should be written so that they can quickly discover those assets upon deployment and start using them.
-

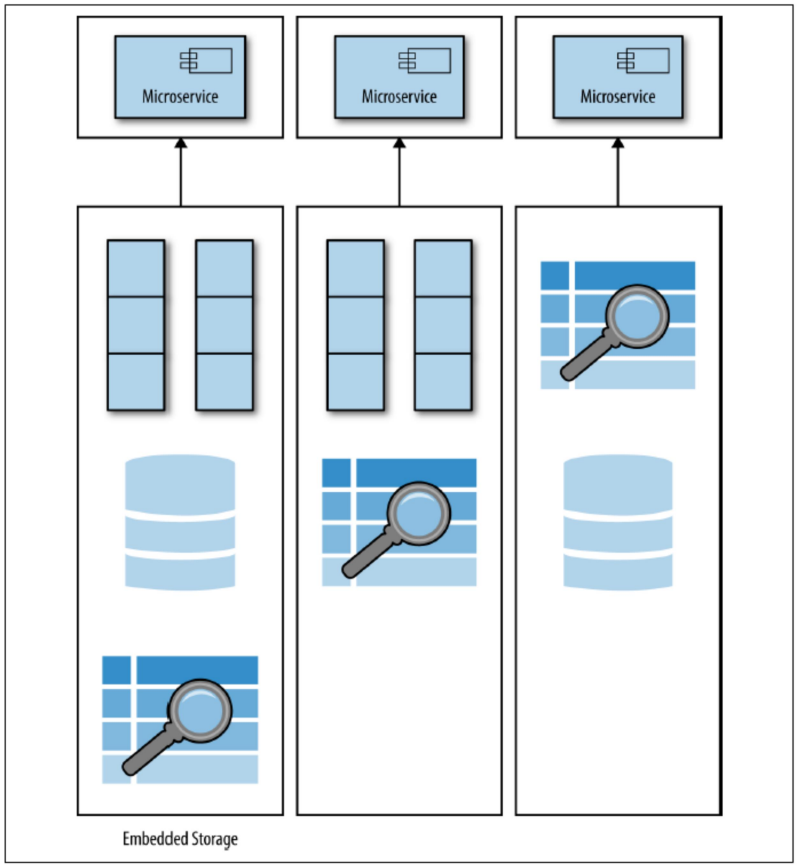


Figure 5-7. Components using fully embedded, isolated dependencies

- **Pragmatic Mobility:**
- 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 (in our earlier metaphor, the hotel provides towels we can use), 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

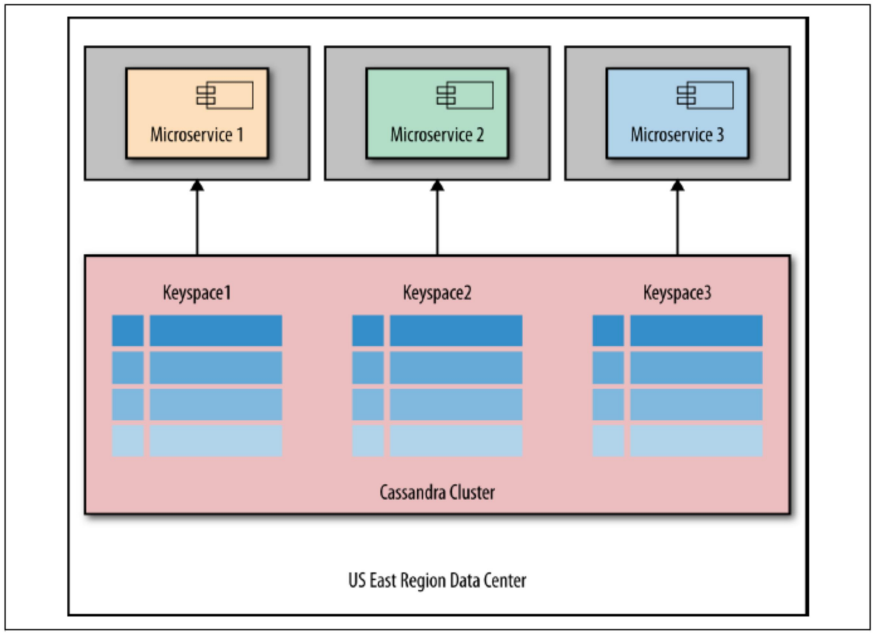


Figure 5-8. Pragmatic approach: Components using a centralized pool of dependencies, without sharing data spaces

- 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.