.NET Microservices

Dr. Ernesto Lee

Sandrilla Thomas

What is a Microservice?

- A *microservice* is a service with one, and only one, very narrowly focused capability that a remote API exposes to the rest of the system.

What is a Microservice - Example

- Receive stock arriving at the warehouse.
- Determine where new stock should be stored.
- Calculate placement routes inside the warehouse for putting stock into the right storage units.
- Assign placement routes to warehouse employees.
- Receive orders.
- Calculate pick routes in the warehouse for a set of orders.
- Assign pick routes to warehouse employees.


Another microservice
Receive stock
Receive stock microservice
Store information about stock in data store
Stock data store

Microservices Architecture

Microservices Architecture

1. **Definition and Scope**: The term "microservices" refers to both individual services and an architectural style for whole systems comprising multiple services.
2. **Comparison with SOA**: Microservices are a lightweight form of service-oriented architecture (SOA), where each service is highly specialized.
3. **Distributed Nature**: Microservices architectures result in distributed systems with potentially a large number of collaborating services.
4. **Growing Popularity**: The adoption of the microservices architectural style is rapidly increasing due to its advantages for server-side software systems.
5. **Key Benefits**: Microservices are malleable, scalable, and robust, offering advantages over traditional SOA and monolithic architectures.
6. **Performance Metrics**: Microservices excel in four key metrics:
    1. Deployment frequency
    2. Lead time for changes
    3. Time to restore service
    4. Change failure rate.

Microservices Characteristics

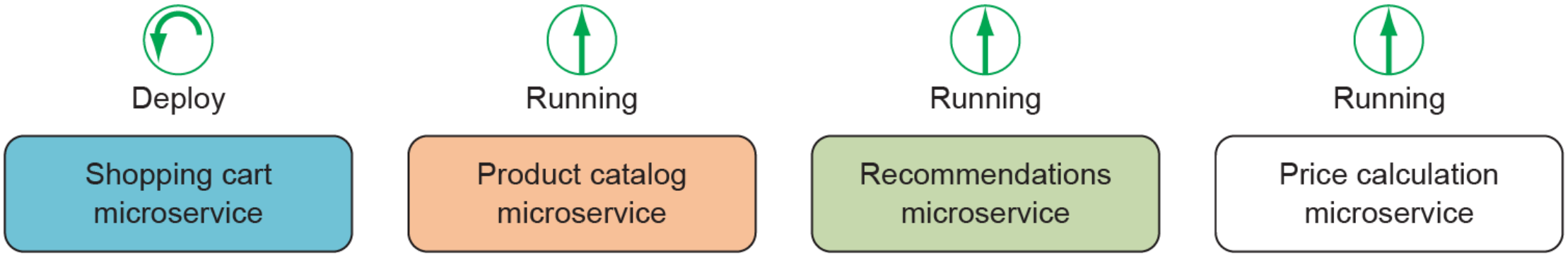
Characteristics of Microservices

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A microservice consists of one or more processes.
- A microservice owns its own data store.
- A small team can maintain a few handfuls of microservices.
- A microservice is replaceable.

Responsible for a single capability

- A microservice has a single responsibility.
- That responsibility is for a capability.

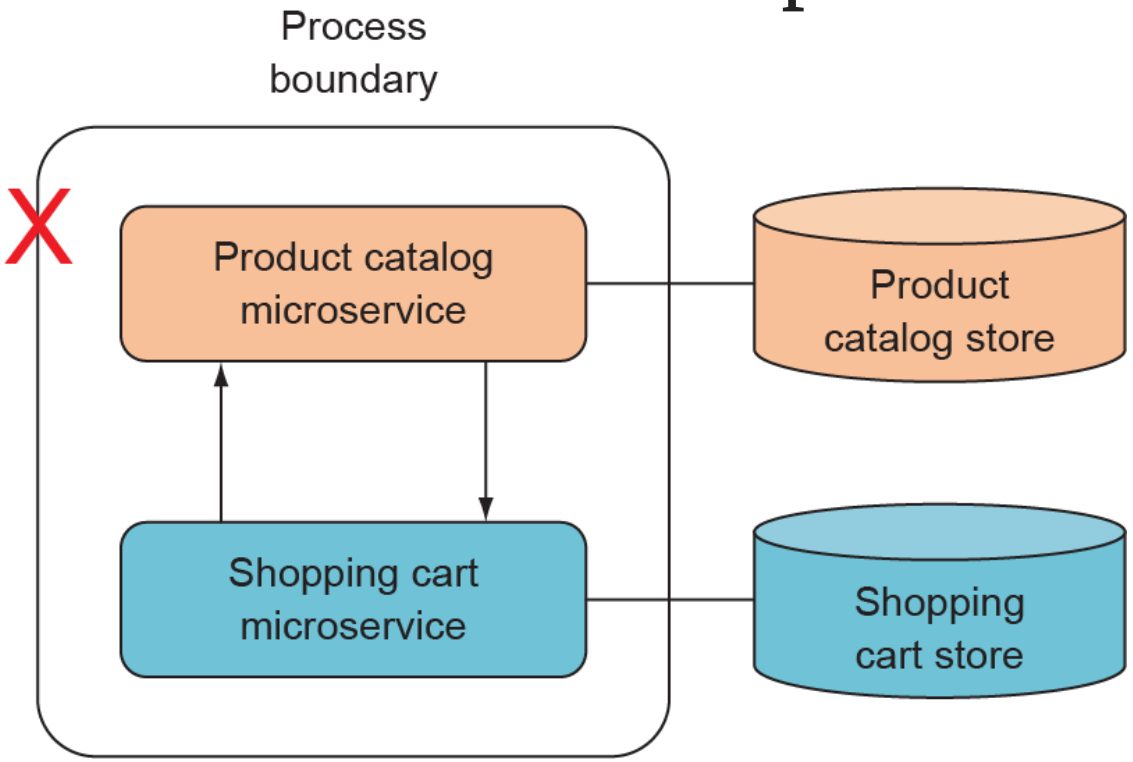
Individually deployable



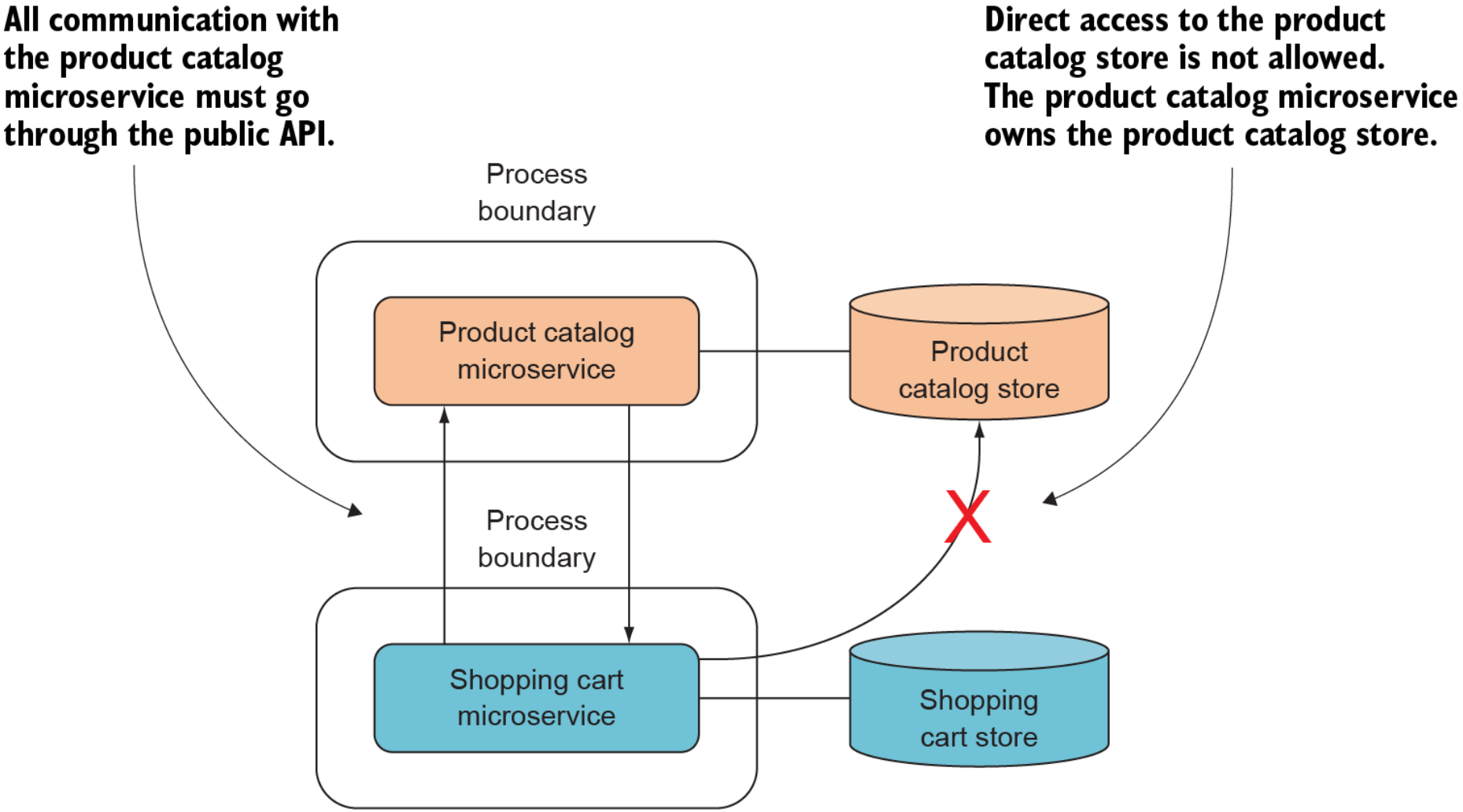
Consists of one or more processes

- Each microservice must run in separate processes from other microservices.
- Each microservice can have more than one process.

Problematic process boundary. Microservices should run in separate processes to avoid coupling.



Owns its own data store



Maintained by a small team

1. **Size Ambiguity**: The term "micro" in microservices doesn't necessarily refer to the number of lines of code or requirements but rather the complexity of the capability it provides.
2. **Maintenance Guideline**: A small team (around five members) should be able to maintain "a few handfuls" of microservices, encompassing development, testing, monitoring, and bug fixes. This can range from 10 to 30 microservices, depending on system maturity and automation.
3. **Team Ownership**: A team should possess a cohesive set of both business and technical capabilities, leading them to own the microservices that represent those functions.

Replaceable

1. **Replaceability Criterion**: A microservice should be small enough to be rewritten from scratch within a regular workflow, ensuring its size doesn't make it prohibitively expensive or time-consuming to replace.
2. **Reasons for Rewriting**: Over time, a microservice may become messy, unmaintainable, or underperforming due to evolving requirements, making a complete rewrite more practical than maintenance.
3. **Informed Rewrites**: If a rewrite is necessary, the team should leverage the knowledge gained from the original implementation and incorporate new requirements to ensure the new microservice is efficient and effective.

Why Microservices

Why Microservices

- Enable continuous delivery
- Allow for an efficient developer workflow because they're highly maintainable
- Are robust by design
- Can scale up or down independently of each other

Enabling continuous delivery

- Can be developed and modified quickly
- Can be comprehensively tested by automated tests
- Can be deployed independently
- Can be operated efficiently

High level of maintainability

- Each well-factored microservice provides *a single capability*. Not two—just one.
- A microservice owns its own data store. No other services can interfere with a microservice's data store. This, combined with the typical size of the codebase for a microservice, means you can understand a complete service all at once.
- Well-written microservices can (and should) be comprehensively covered by automated tests.

Robust and scalable

1. **Individual Scalability**: In a microservices architecture, each service can be scaled independently, allowing for targeted scaling based on specific bottlenecks.
2. **Asynchronous Collaboration**: Microservices prioritize asynchronous, event-based interactions, reducing the dependencies and bottlenecks associated with synchronous communication.
3. **Fault Tolerance and High Availability**: Emphasizing fault tolerance in synchronous interactions ensures that systems built with microservices are both highly available and scalable.

Costs and downsides of microservices

1. **Challenges of Distributed Systems**: Microservice architectures, being distributed systems, are inherently more complex, difficult to reason about, and test compared to monolithic systems. Moreover, communication across them is substantially slower.
2. **Operational Complexity**: With a microservices setup, each service requires its own development, deployment, and management in production, leading to numerous deployments and intricate production configurations.
3. **Refactoring Difficulties**: Since each microservice has its own codebase, moving code between services is cumbersome, necessitating careful scoping of each microservice from the outset.
4. **Weighing Costs vs. Complexity**: Before adopting microservices, it's crucial to evaluate if the system's complexity warrants the overhead and challenges associated with a microservices architecture.

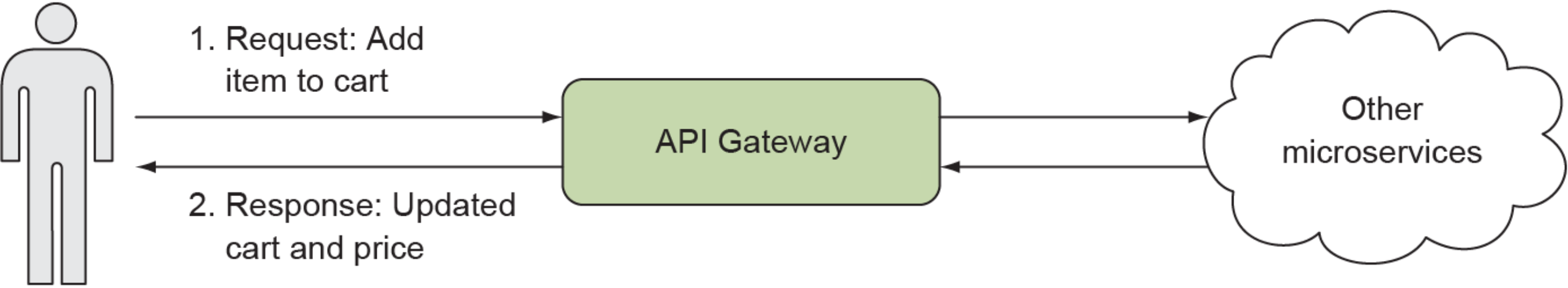
Greenfield vs. brownfield

1. Microservices Decision: Deciding between introducing microservices at the start or only for large existing systems is a common dilemma.
2. Monolithic System Evolution: Many systems originate as small applications but evolve into large monolithic structures over time, leading to inherent challenges.
3. Challenges of Monoliths:
    1. High coupling throughout the codebase.
    2. Hidden couplings, like implicit knowledge in code about data formatting or database usage.
    3. Deployment complexities, involving multiple stakeholders and potential downtime.
    4. Over-engineering of simpler components due to a one-size-fits-all architectural approach.
4. Origins of Microservices: The microservices architecture emerged as a solution to the problems faced by monolithic systems.
5. Natural Progression: Breaking down components of a monolith repeatedly can eventually lead to the creation of microservices, allowing for more manageable and modular components.

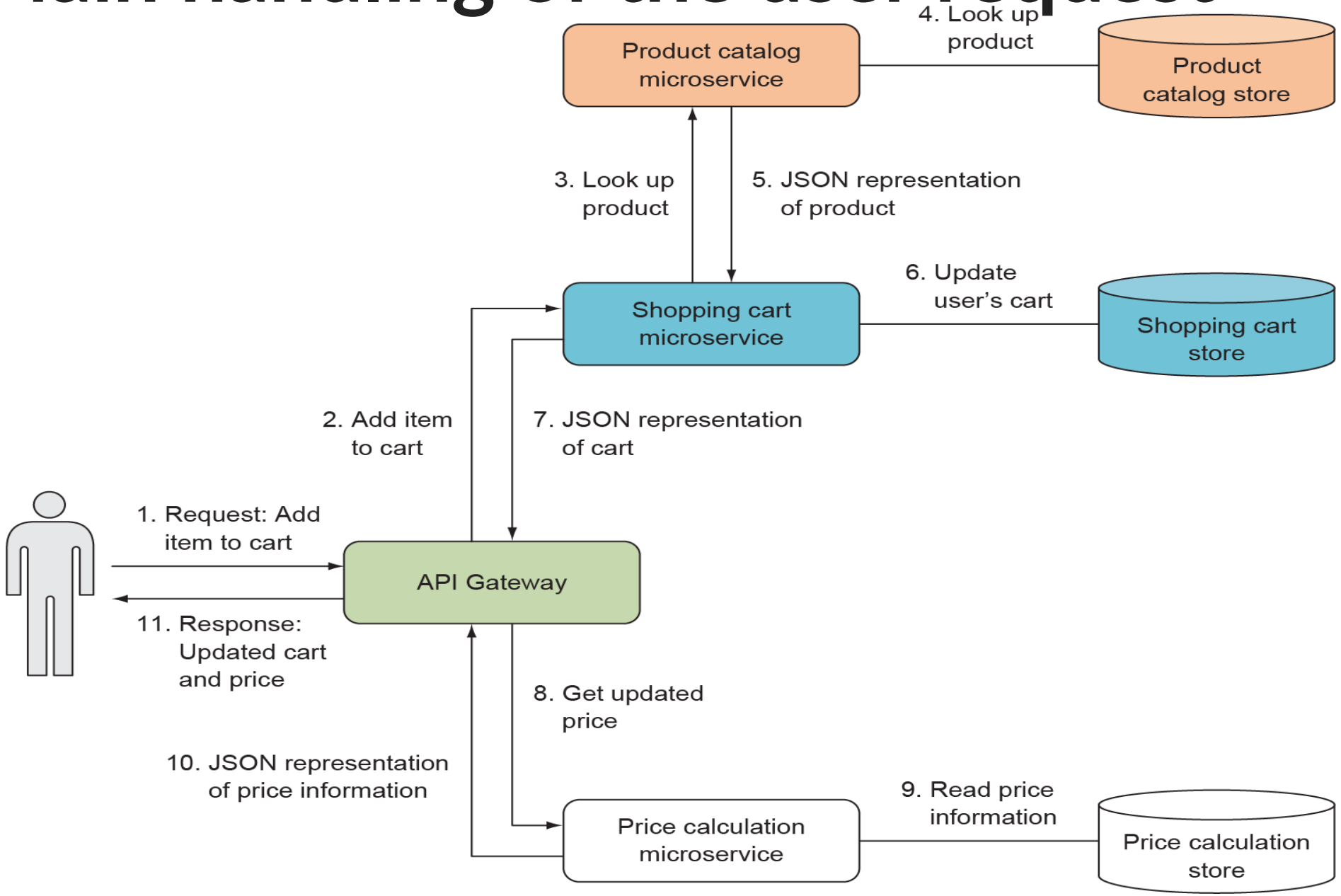
Code reuse

1. **Challenges of Code Reuse**:
    1. Extracting code into a library introduces an additional dependency, complicating service understanding and maintenance.
    2. Code in shared libraries must cater to multiple use cases, increasing development effort.
    3. Shared libraries can inadvertently introduce harmful coupling between services, leading to complexities during updates and potential errors.
    4. Business code should generally not be reused across microservices to avoid unnecessary and detrimental coupling.
2. **Business vs. Infrastructure Code**: While business code reuse can introduce harmful coupling, there's merit in reusing infrastructure code addressing technical concerns.
3. **Maintaining Service Purity**: To ensure a microservice remains focused on its core capability, it's often better to create a new service than to add functionalities to an existing one.
4. **Efficient Service Creation**: Reusable platforms addressing common technical concerns can streamline the creation of new services, ensuring consistency and reducing the development effort.

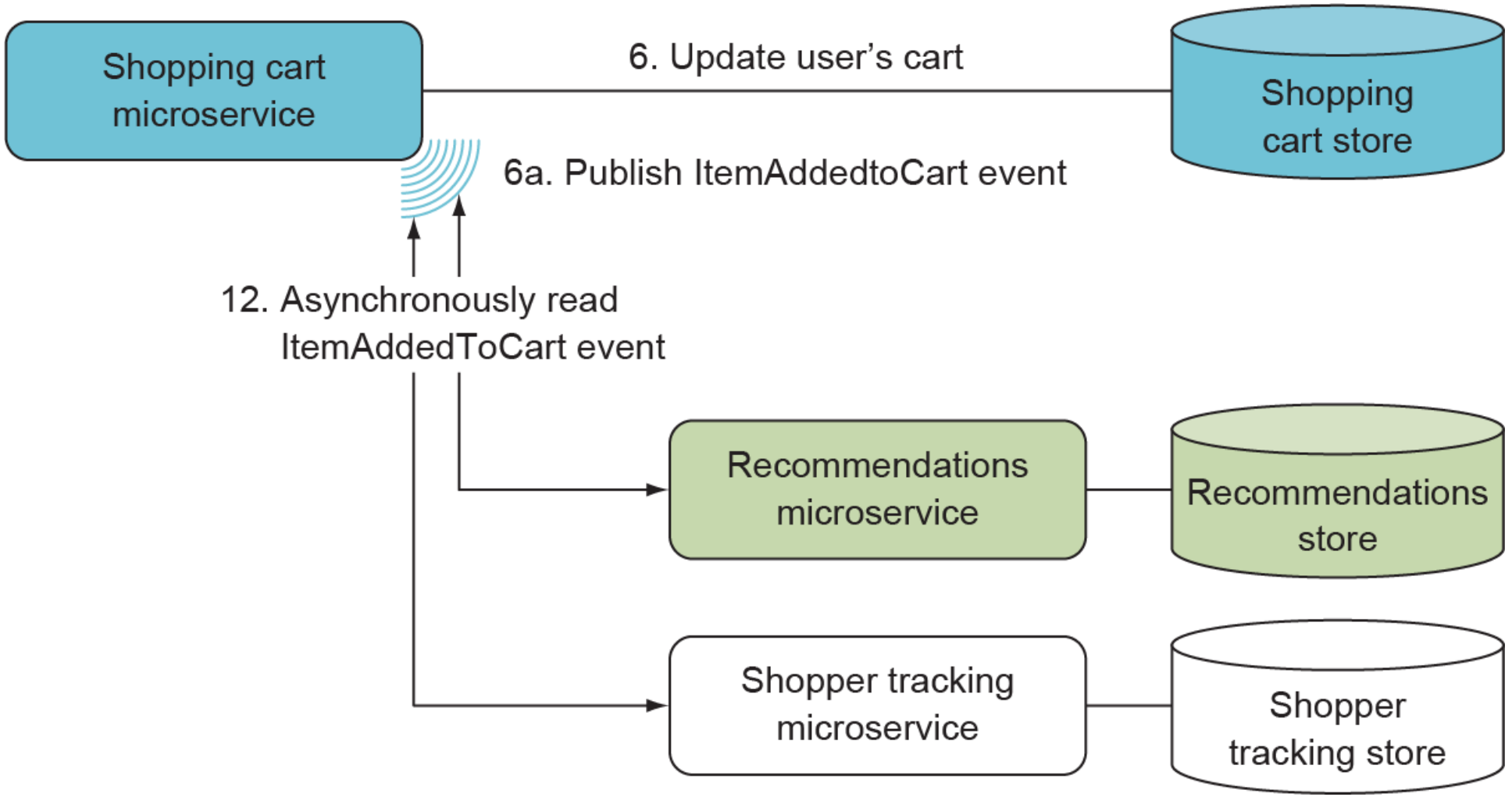
Serving a user request: An example of how microservices work in concert



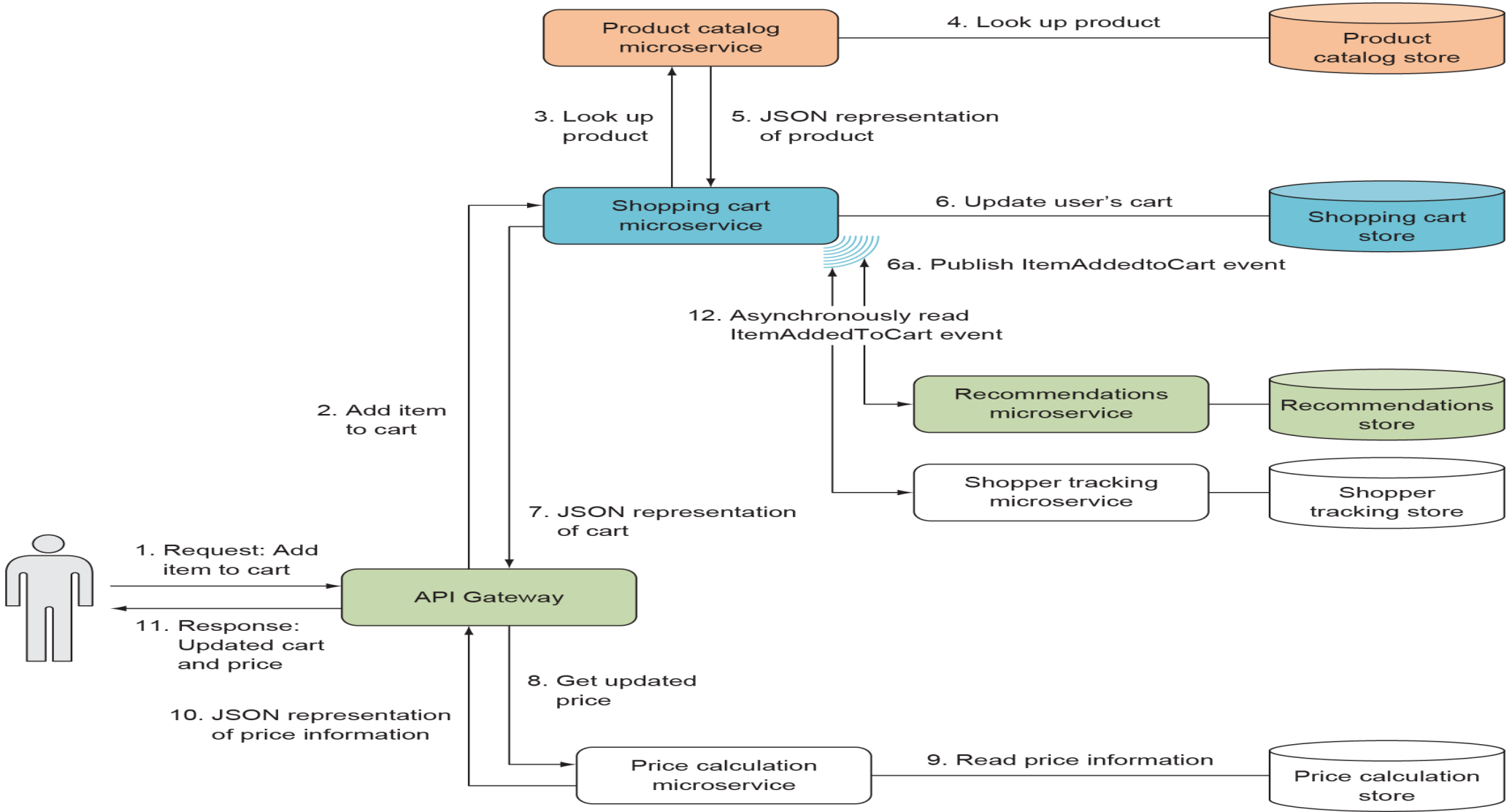
Main handling of the user request



Side effects of the user request

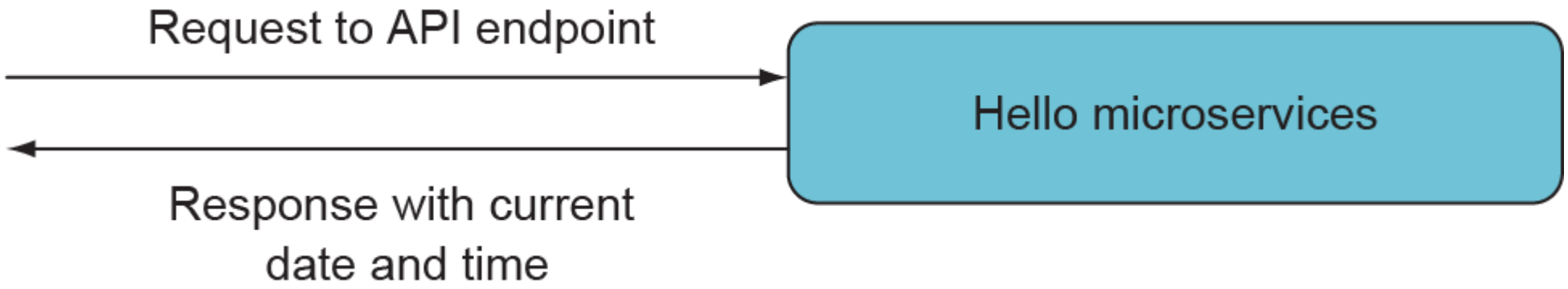


The complete picture



A .NET microservices technology stack

Let’s see some code!



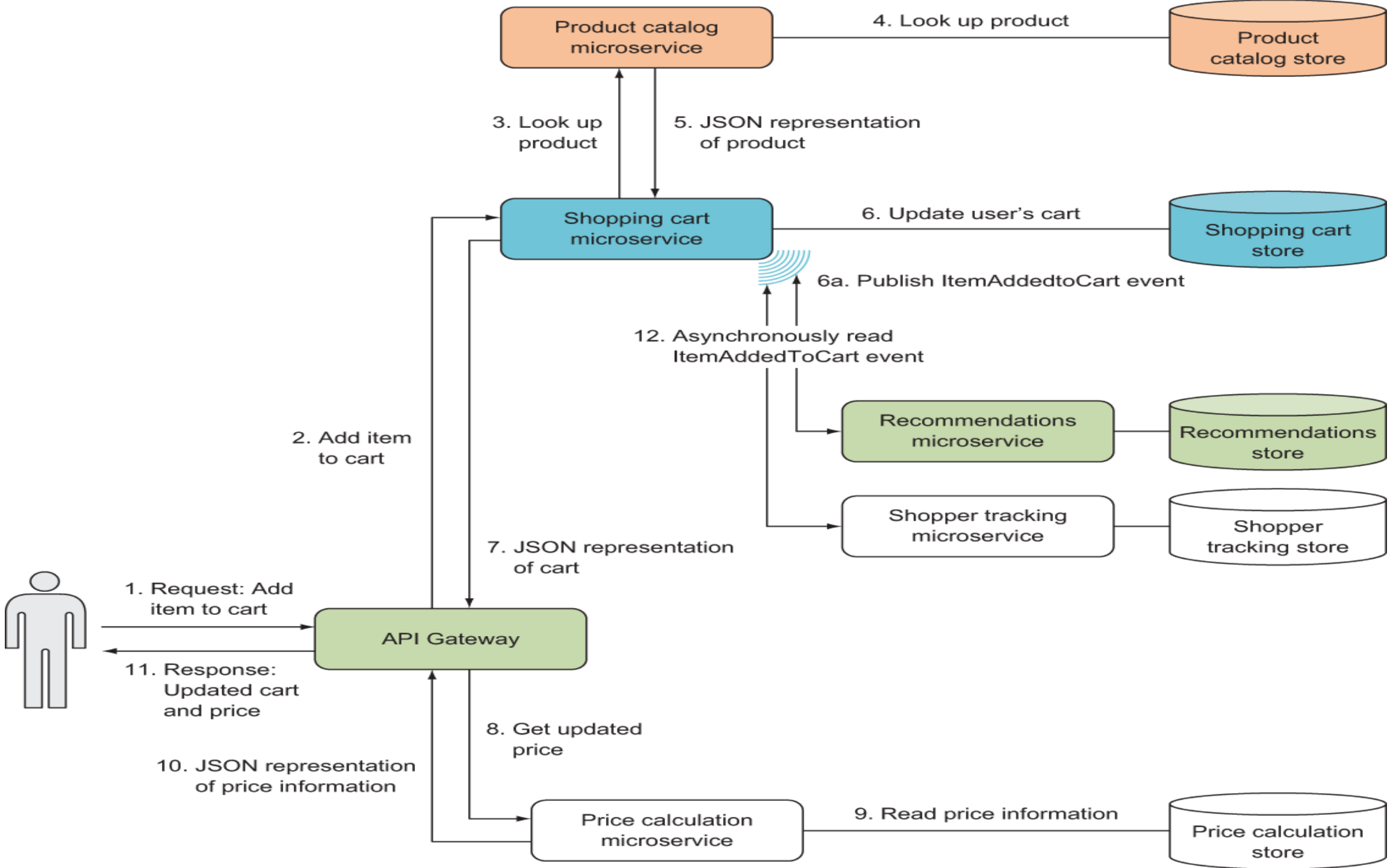
To implement this example, you'll follow these three steps:

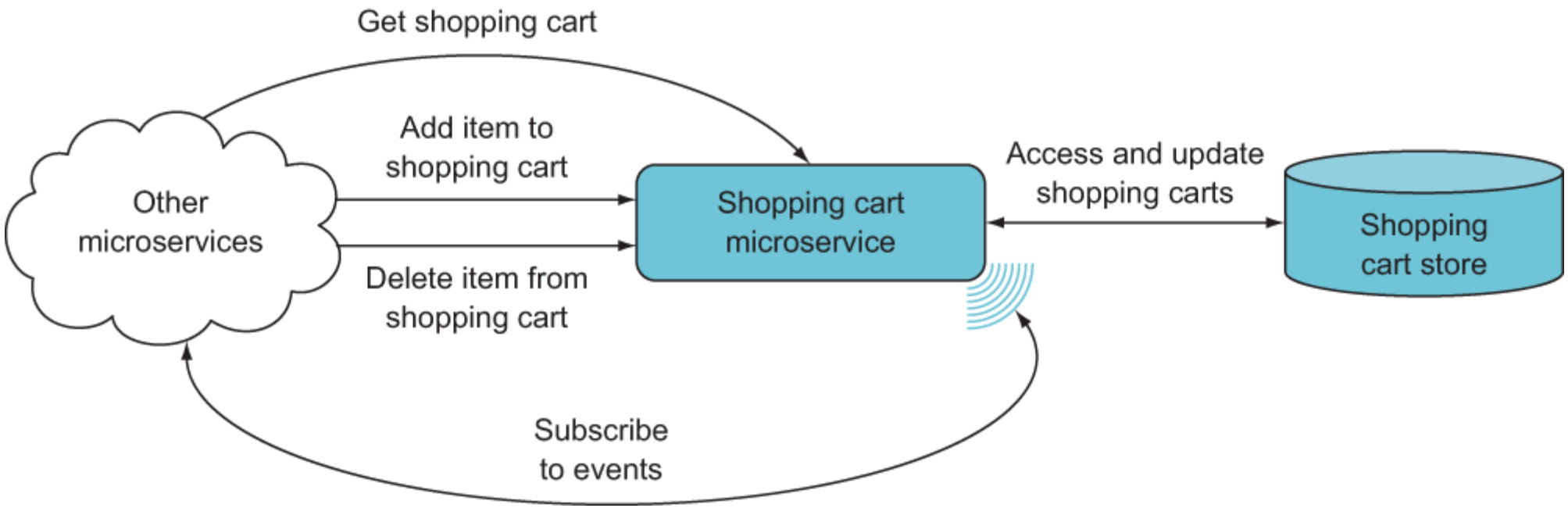
1. Create an empty ASP.NET application.
2. Add ASP.NET MVC to the application.
3. Add an MVC controller with an implementation of the endpoint.

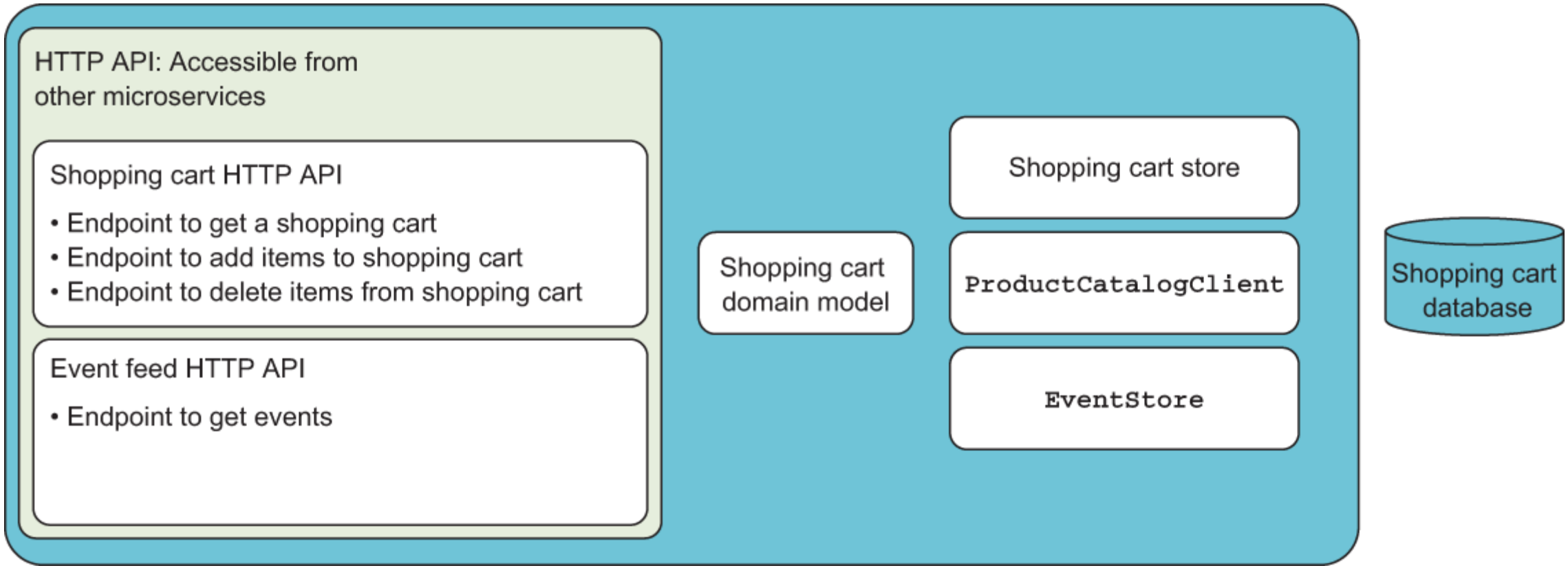
Summary

1. **Definition and Scope**:
    1. "Microservices" refers to both an architectural style and individual units within a system.
    2. It's a specialized form of SOA, focusing on delivering a single business capability.
2. **Characteristics of a Microservice**:
    1. Offers one capability.
    2. Independently deployable.
    3. Runs in separate processes.
    4. Manages its own data.
    5. A small team can manage multiple microservices.
    6. Can be rewritten quickly if necessary.
3. **Relation to Continuous Delivery**:
    1. Simplifies the continuous delivery process.
    2. Facilitates reliable and rapid deployment.
4. **Benefits of Microservices**:
    1. Promotes scalability and resilience.
    2. Offers flexibility and adaptability to business needs.
    3. Each unit is maintainable and can be quickly developed.
    4. Microservices work collaboratively to deliver user functionality.
5. **Interaction Mechanisms**:
    1. Provides a remote public API.
    2. Can publish event feeds for other services to subscribe to, ensuring fast asynchronous reactions.
6. **Implementation Tools**:
    1. ASP.NET is a viable platform for creating microservices.
    2. Kubernetes is the chosen container orchestrator for the course.
    3. Microservices typically serve data (like JSON) instead of HTML. Tools like REST client, Postman, and Fiddler are useful for testing these endpoints.

A Basic Service


Product catalog microservice
4. Look up product
Product catalog store
3. Look up product
5. JSON representation of product
Shopping cart microservice
6. Update user's cart
Shopping cart store
6a. Publish ItemAddedtoCart event
12. Asynchronously read ItemAddedToCart event
Recommendations microservice
Recommendations store
2. Add item to cart
Shopper tracking microservice
Shopper tracking store
7. JSON representation of cart
1. Request: Add item to cart
API Gateway
11. Response: Updated cart and price
8. Get updated price
10. JSON representation of price information
Price calculation microservice
9. Read price information
Price calculation store


Get shopping cart
Add item to
shopping cart
Other
microservices
Delete item from
shopping cart
Shopping cart
microservice
Access and update
shopping carts
Shopping
cart store
Subscribe
to events


HTTP API: Accessible from other microservices
Shopping cart HTTP API
• Endpoint to get a shopping cart
• Endpoint to add items to shopping cart
• Endpoint to delete items from shopping cart
Event feed HTTP API
• Endpoint to get events
Shopping cart domain model
Shopping cart store
ProductCatalogClient
EventStore
Shopping cart database

Implementing the Shopping Cart microservice

- System.Net.Http.HttpClient
- Polly
- Scrutor

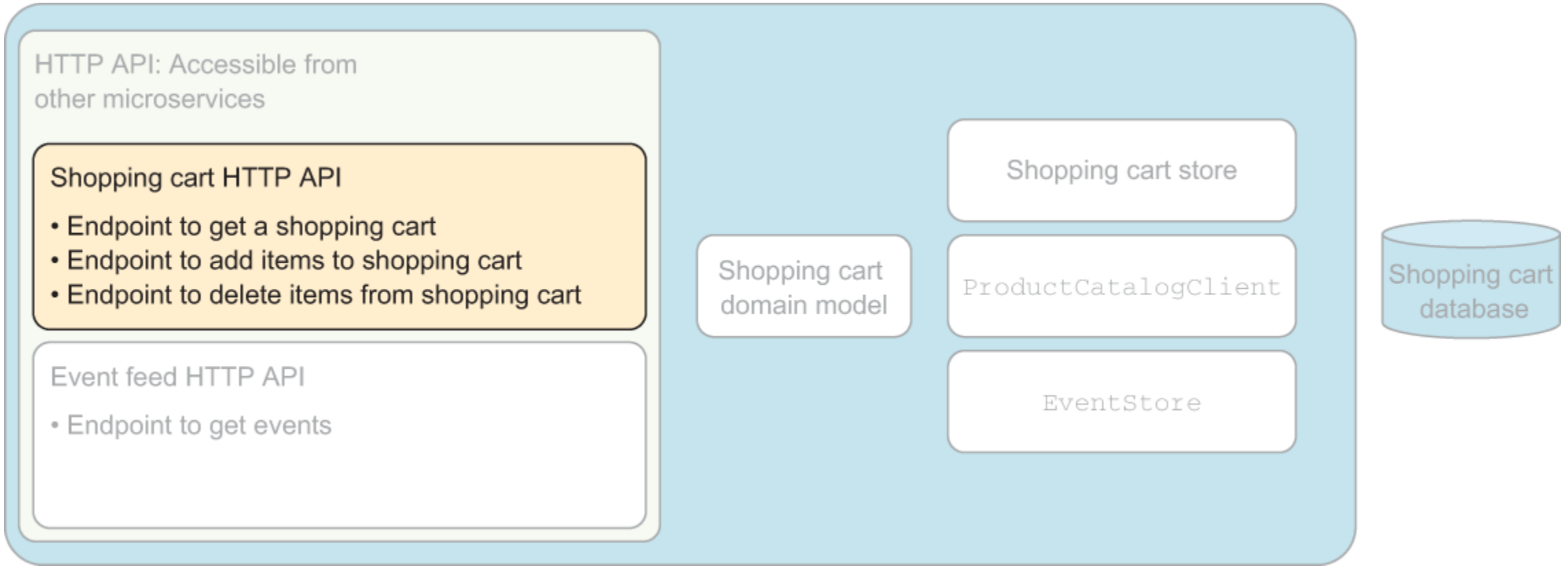
dotnet cli

- dotnet new web –n ShoppingCart

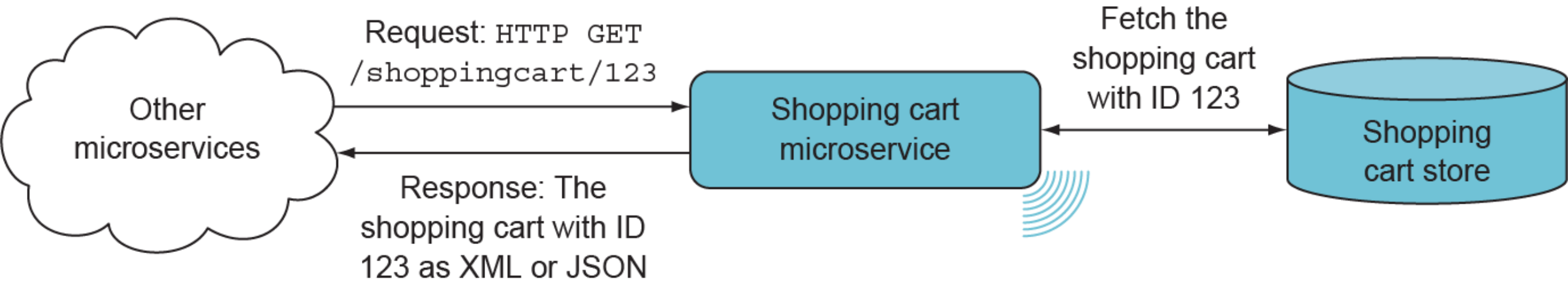
Create an Empty Project

```
namespace ShoppingCart
{
 using Microsoft.AspNetCore.Builder;
 using Microsoft.Extensions.DependencyInjection;

 public class Startup
 {
  public void ConfigureServices(IServiceCollection services)
  {
   services.AddControllers();
  }

  public void Configure(IApplicationBuilder app)
  {
   app.UseHttpsRedirection();
   app.UseRouting();
   app.UseEndpoints(endpoints =>
    endpoints.MapControllers());
  }
 }
```

The Shopping Cart microservice's API for other services



Getting a shopping cart



Getting a shopping cart

For example, the API Gateway may need the shopping cart for a user with ID 123. To get that, it sends this HTTP request:

```
HTTP GET /shoppingcart/123 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
```

```
ShoppingCart
│   appsettings.Development.json
│   appsettings.json
│   Program.cs
│   ShoppingCart.csproj
│   Startup.cs
│
└── ShoppingCart
        ShoppingCart.cs
        ShoppingCartController.cs
        ShoppingCartStore.cs
```

- The Shopping Cart domain model is simple and looks like this:

```
namespace ShoppingCart.ShoppingCart
{
  using System.Collections.Generic;
  using System.Linq;

  public class ShoppingCart
  {
    private readonly HashSet<ShoppingCartItem> items = new();

    public int UserId { get; }
    public IEnumerable<ShoppingCartItem> Items => this.items;

    public ShoppingCart(int userId) => this.UserId = userId;

    public void AddItems(IEnumerable<ShoppingCartItem> shoppingCartItems)
    {
      foreach (var item in shoppingCartItems)
        this.items.Add(item);
    }

    public void RemoveItems(int[] productCatalogueIds) =>
      this.items.RemoveWhere(i => productCatalogueIds.Contains(
          i.ProductCatalogueId));
  }

  public record ShoppingCartItem(
    int ProductCatalogueId,
    string ProductName,
    string Description,
    Money Price)
  {
    public virtual bool Equals(ShoppingCartItem? obj) =>
      obj != null && this.ProductCatalogueId.Equals(obj.ProductCatalogueId);

    public override int GetHashCode() =>
      this.ProductCatalogueId.GetHashCode();
  }

  public record Money(string Currency, decimal Amount);
}
```

```
namespace ShoppingCart.Shoppingcart
{
  using Microsoft.AspNetCore.Mvc;
  using ShoppingCart;

  [Route("/shoppingcart")]
  public class ShoppingCartController : ControllerBase
  {
    private readonly IShoppingCartStore shoppingCartStore;

    public ShoppingCartController(IShoppingCartStore shoppingCartStore)
    {
      this.shoppingCartStore = shoppingCartStore;
    }

    [HttpGet("{userId:int}")]
    public ShoppingCart Get(int userId) =>
      this.shoppingCartStore.Get(userId);
  }
}
```

- The HttpGet attribute is followed by a method:

```
public ShoppingCart Get(int userId) =>
    this.shoppingCartStore.Get(userId);
```

- The action method uses a shoppingCartStore object that the ShoppingCartController constructor takes as an argument and assigns to an instance variable:

```
public ShoppingCartController(IShoppingCartStore shoppingCartStore)
{
  this.shoppingCartStore = shoppingCartStore;
}
```

First, we add Scrutor to the shopping cart project by running this dotnet command:

```
PS> dotnet add package scrutor
```

- This installs the Scrutor NuGet package, which we can see in the shopping cart project file—ShoppingCart.csproj—where there now is a package reference to Scrutor:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Scrutor" Version="3.1.0" />
  </ItemGroup>

</Project>
```

- With Scrutor installed, we can add code to the ConfigureServices method in the Startup class, so it becomes

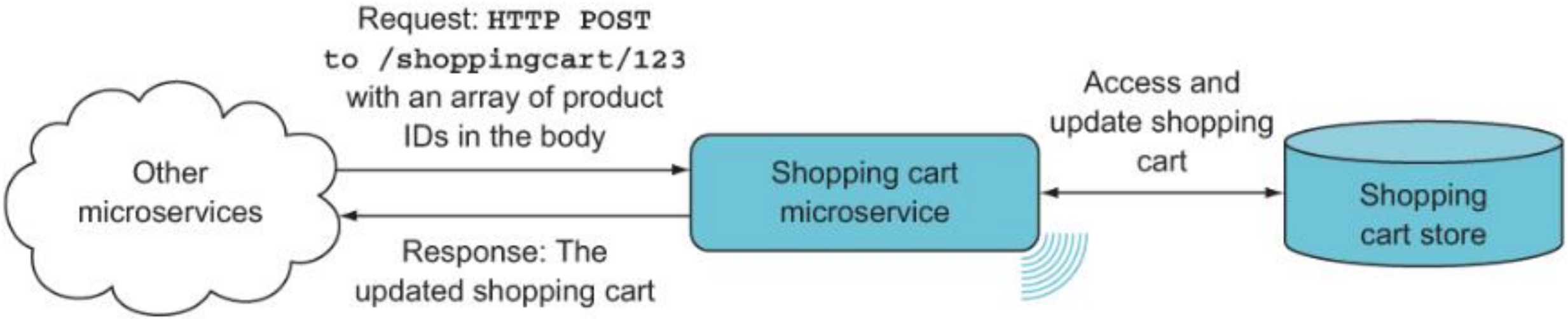
```
public void ConfigureServices(IServiceCollection services)
   {
     services.AddControllers();
     services.Scan(selector =>
       selector
         .FromAssemblyOf<Startup>()
         .AddClasses()
         .AsImplementedInterfaces());
   }
```

```
public interface IShoppingCartStore
 {
  ShoppingCart Get(int userId);
  void Save(ShoppingCart shoppingCart);
 }

 public class ShoppingCartStore : IShoppingCartStore
 {
  private static readonly Dictionary<int, ShoppingCart>
    Database = new Dictionary<int, ShoppingCart>();

  public ShoppingCart Get(int userId) =>
    Database.ContainsKey(userId)
    ? Database[userId]
    : new ShoppingCart(userId);

  public void Save(ShoppingCart shoppingCart) =>
    Database[shoppingCart.UserId] = shoppingCart;
 }
```

- Returning attention to the ShoppingCartController, the action method Get returns a ShoppingCart object that it gets back from shoppingCartStore:

```
this.shoppingCartStore.Get(userId);
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

539
{
    "userId": 42,
    "items": [
        {
            "productcatalogId": 1,
            "productName": "Basic t-shirt",
            "description": "a quiet t-shirt",
            "price": {
                "currency": "eur",
                "amount": 40
            }
        },
        {
            "productcatalogId": 2,
            "productName": "Fancy shirt",
            "description": "a loud t-shirt",
            "price": {
                "currency": "eur",
                "amount": 50
            }
        }
    ]
}
```


Request: HTTP POST
to /shoppingcart/123
with an array of product
IDs in the body
Other
microservices
Response: The
updated shopping cart
Shopping cart
microservice
Access and
update shopping
cart
Shopping
cart store

Adding two items to a shopping cart

```
POST /shoppingcart/123/items HTTP/1.1
  Host: shoppingcart.my.company.com
  Accept: application/json
  Content-Type: application/json

  [1, 2]
```

```
[Route("/shoppingcart")]
public class ShoppingCartController : Controller
{
  private readonly IShoppingCartStore shoppingCartStore;
  private readonly IProductCatalogClient productCatalogClient;
  private readonly IEventStore eventStore;

  public ShoppingCartController(
    IShoppingCartStore shoppingCartStore,
    IProductCatalogClient productCatalogClient,
    IEventStore eventStore)
  {
    this.shoppingCartStore = shoppingCartStore;
    this.productCatalogClient = productCatalogClient;
    this.eventStore = eventStore;
  }

  [HttpGet("{userId:int}")]
  public ShoppingCart Get(int userId) =>
    this.shoppingCartStore.Get(userId);

  [HttpPost("{userId:int}/items")]
  public async Task<ShoppingCart> Post(
    int userId,
    [FromBody] int[] productIds)
  {
    var shoppingCart = shoppingCartStore.Get(userId);
    var shoppingCartItems =
      await this.productcatalogClient
        .GetShoppingCartItems(productIds);
    shoppingCart.AddItems(shoppingCartItems, eventStore);
    shoppingCartStore.Save(shoppingCart);
    return shoppingCart;
  }
}
```

- Second, the body of the request contains a JSON array of product IDs. These are the items that should be added to the shopping cart. The action uses MVC model binding to read these into a C# array:

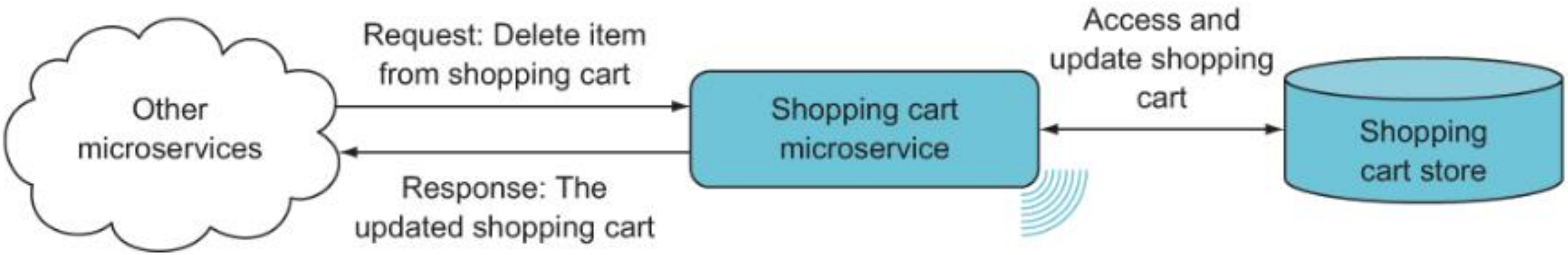
```
[FromBody] int[] productIds
```

- You once again rely on ASP.NET to provide them through constructor arguments.

```
public ShoppingCartController(
  IShoppingCartStore shoppingCartStore,
  IProductCatalogClient productCatalogClient,
  IEventStore eventStore)
```

- The remainder of the method is queued up for execution when the awaitable—in this case, the Task returned from Task.Delay(1000)—completes. When the awaitable completes, the rest of the method is executed, possibly on a new thread but with same state as before the await re-established.

- The current thread of execution returns from the async method and continues in the caller.


Other microservices
Request: Delete item from shopping cart
Response: The updated shopping cart
Shopping cart microservice
Access and update shopping cart
Shopping cart store

```
[HttpDelete("{userid:int}/items")]
public ShoppingCart Delete(
  int userId,
  [FromBody] int[] productIds)
{
  var shoppingCart =
    this.shoppingCartStore.Get(userId);
  shoppingCart.RemoveItems(
    productIds,
    this.eventStore);
  this.shoppingCartStore.Save(shoppingCart);
  return shoppingCart;
}
```

Fetching product information

You need to follow these three steps to implement the HTTP request to the product catalog microservice:

- Implement the HTTP GET request.
- Parse the response from the endpoint at the product catalog microservice and translate it to the domain of the shopping cart microservice.
- Implement a policy for handling failed requests to the product catalog microservice.

Implementing the HTTP GET

- For example, the following request fetches the information for product IDs 1 and 2:

```
HTTP GET /products?productIds=[1,2] HTTP/1.1
Host: productcatalog.my.company.com
Accept: application/json
```

```
public class ProductCatalogClient : IProductCatalogClient
  {
    private readonly HttpClient client;
    private static string productCatalogBaseUrl =
      @"https://git.io/JeHiE";
    private static string getProductPathTemplate = "?productIds=[{0}]";

    public ProductCatalogClient(HttpClient client)
    {
      client.BaseAddress =
          new Uri(productCatalogBaseUrl);
      client
        .DefaultRequestHeaders
        .Accept
          .Add(new MediaTypeWithQualityHeaderValue("application/json"));
      this.client = client;
    }

    public async Task<IEnumerable<ShoppingCartItem>>
      GetShoppingCartItems(int[] productCatalogIds)
    {
      ....
    }

    private async Task<HttpResponseMessage>
      RequestProductFromProductCatalog(int[] productCatalogIds)
    {
      var productsResource =
        string.Format(getProductPathTemplate,
          string.Join(",", productCatalogIds));
      return await
        this.client.GetAsync(productsResource);
    }
  }
```

- For ASP.NET to be able to inject the HttpClient in ProductCatalogClient we need to register ProductCatalogClient as typed http client, which we do by adding this to the ConfigureServices method in Startup:

```
services.AddHttpClient<IProductCatalogClient, ProductCatalogClient>();
```

Parsing the product response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

543
[
  {
    "productId": "1",
    "productName": "Basic t-shirt",
    "productDescription": "a quiet t-shirt",
    "price": { "amount" : 40, "currency": "eur" },
    "attributes" : [
    {
      "sizes": [ "s", "m", "l"],
      "colors": ["red", "blue", "green"]
    }]
  },
  {
    "productId": "2",
    "productName": "Fancy shirt",
    "productDescription": "a loud t-shirt",
    "price": { "amount" : 50, "currency": "eur" },
    "attributes" : [
    {
      "sizes": [ "s", "m", "l", "xl"],
      "colors": ["ALL", "Batique"]
    }]
  }
]
```

```
private static async Task<IEnumerable<ShoppingCartItem>>
      ConvertToShoppingCartItems(HttpResponseMessage response)
    {
       response.EnsureSuccessStatusCode();
      var products = await
        JsonSerializer.DeserializeAsync<List<ProductCatalogProduct>>(
          await response.Content.ReadAsStreamAsync(),
          new JsonSerializerOptions
          {
             PropertyNameCaseInsensitive = true
          }) ?? new();
      return products
        .Select(p =>
          new ShoppingCartItem(
            p.ProductId,
            p.ProductName,
            p.ProductDescription,
            p.Price
      ));
    }

    private record ProductCatalogProduct(
      int ProductId,
      string ProductName,
      string ProductDescription,
      Money Price);
```

The following listing combines the code that requests the product information and the code that parses the response.

```
public async Task<IEnumerable<ShoppingCartItem>>
  GetShoppingCartItems(int[] productCatalogIds)
{
  using var response =
    await RequestProductFromProductCatalogue(productCatalogIds);
  return await ConvertToShoppingCartItems(response);
}
```

Adding a failure-handling policy

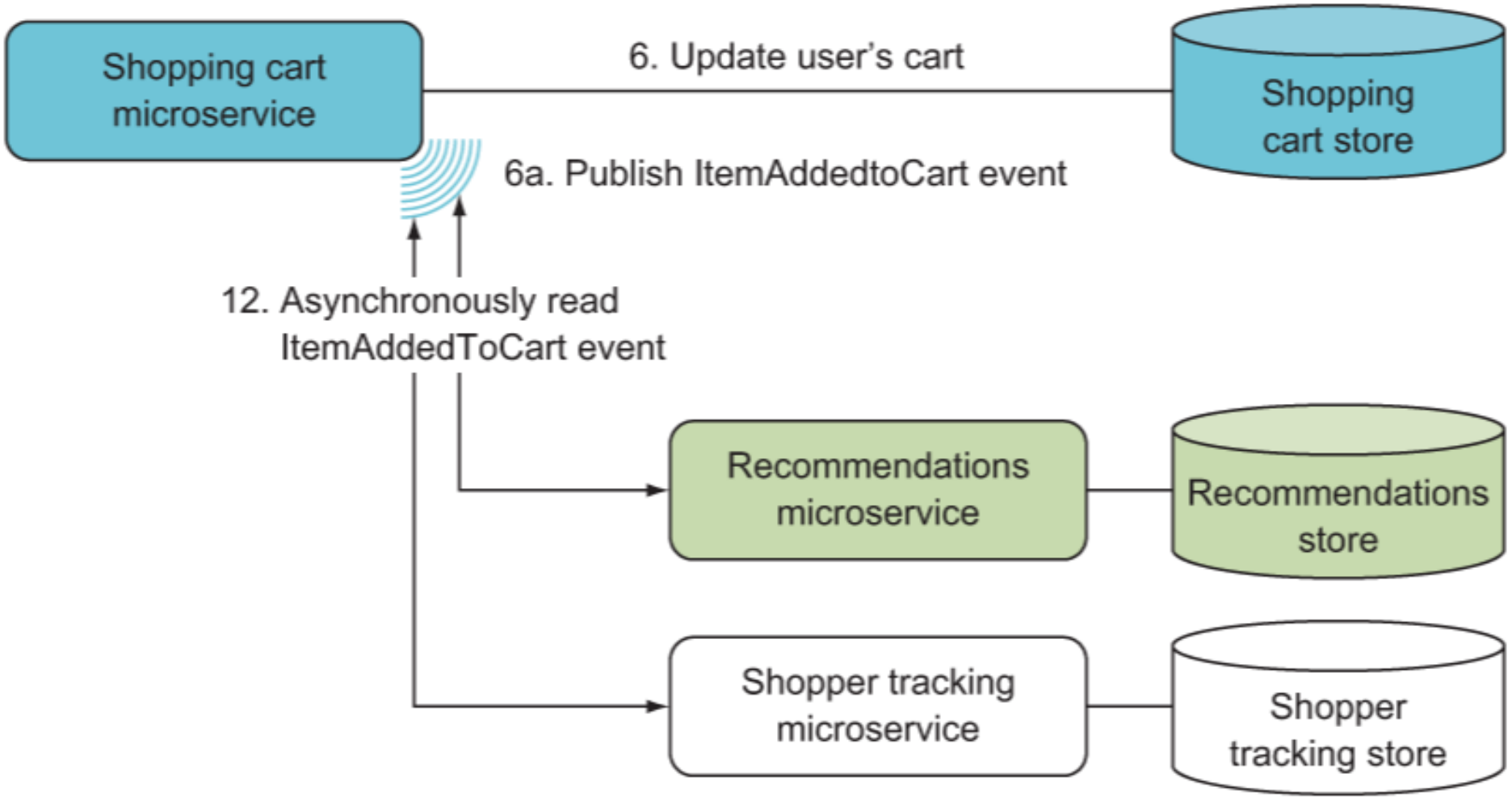
Caching product information has some significant advantages:

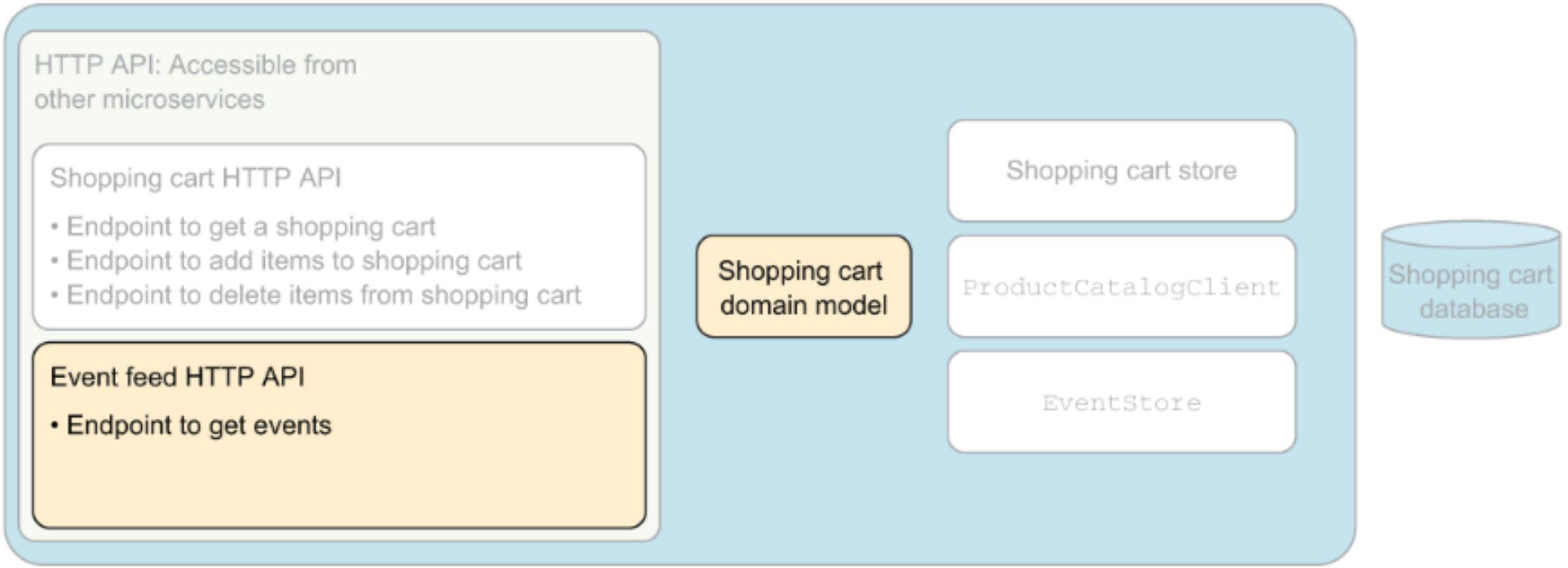
- It makes the shopping cart more resilient to failures in product catalog.
- The shopping cart microservice will perform better when the product information is present in the cache.
- Fewer calls made from the shopping cart microservice mean less stress is put on the product catalog microservice.

- As you can see in the following listing, Polly's API and integration with ASP.NET makes both these steps easy. Replace the current registration of ProductCatalogClient in Startup with this:

```
services.AddHttpClient<IProductCatalogClient, ProductCatalogClient>()
    .AddTransientHttpErrorPolicy(p =>
      p.WaitAndRetryAsync(
        3,
        attempt => TimeSpan.FromMilliseconds(100*Math.Pow(2, attempt))));
```

Implementing a basic event feed




HTTP API: Accessible from other microservices
Shopping cart HTTP API
• Endpoint to get a shopping cart
• Endpoint to add items to shopping cart
• Endpoint to delete items from shopping cart
Event feed HTTP API
• Endpoint to get events
Shopping cart domain model
Shopping cart store
ProductCatalogClient
EventStore
Shopping cart database

Implementing the event feed involves these steps:

- *Raise events*.
- *Store events*.
- *Publish events*.

- The first file contains the model type for events, shown in the following listing.

```
namespace ShoppingCart.EventFeed
{
  using System;

  public record Event(
    long SequenceNumber,
    DateTimeOffset OccuredAt,
    string Name,
    object Content);
}
```

Using this interface, the ShoppingCart domain object can raise events, as shown next.

```
public void AddItems(
      IEnumerable<ShoppingCartItem> shoppingCartItems,
      IEventStore eventStore)
{
  foreach (var item in shoppingCartItems)
    if (this.items.Add(item))
      eventStore.Raise(
        "ShoppingCartItemAdded",
        new { UserId, item });
}
```

Storing an event

```
public void Raise(string eventName, object content)
{
  var seqNumber = database.NextSequenceNumber();
  database.Add(
    new Event(
      seqNumber,
      DateTimeOffset.UtcNow,
      eventName,
      content));
}
```

A simple event feed

- A subscriber can, for example, issue the following request to get all events newer than event number 100:

```
GET /events?start=100 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
```

- Or, if the subscriber wants to limit the number of incoming events per call, it can add an end argument to the request:

```
GET /events?start=100&end=200 HTTP/1.1
Host: shoppingcart.my.company.com
Accept: application/json
```

```
namespace ShoppingCart.EventFeed
{
  using System.Linq;
  using Microsoft.AspNetCore.Mvc;

  [Route("/events")]
  public class EventFeedController : Controller
  {
    private readonly IEventStore eventStore;

    public EventFeedController(IEventStore eventStore) =>
      this.eventStore = eventStore;

    [HttpGet("")]
    public Event[] Get(
      [FromQuery] long start,
      [FromQuery] long end = long.MaxValue)
    =>
      this.eventStore
        .GetEvents(start, end)
        .ToArray();
  }
}
```

- The following simple implementation illustrates it well.

```
public IEnumerable<Event> GetEvents(
  long firstEventSequenceNumber,
  long lastEventSequenceNumber)  =>
    database
      .Where(e =>
        e.SequenceNumber >= firstEventSequenceNumber &&
        e.SequenceNumber <= lastEventSequenceNumber)
      .OrderBy(e => e.SequenceNumber);
```

Running the code

- Now that all the code for the shopping cart microservice is in place, you can run it.

- You can test all the endpoints with RestClient or a similar tool.

Summary

- Implementing a complete microservice doesn’t take much code.
- The Polly library is useful for implementing failure-handling policies and wrapping them around remote calls.
- You should always expect that other microservices may be down. To prevent errors from propagating, each remote call should be wrapped in a policy for handling failure.

Deploying a microservice to Kubernetes

Introduction

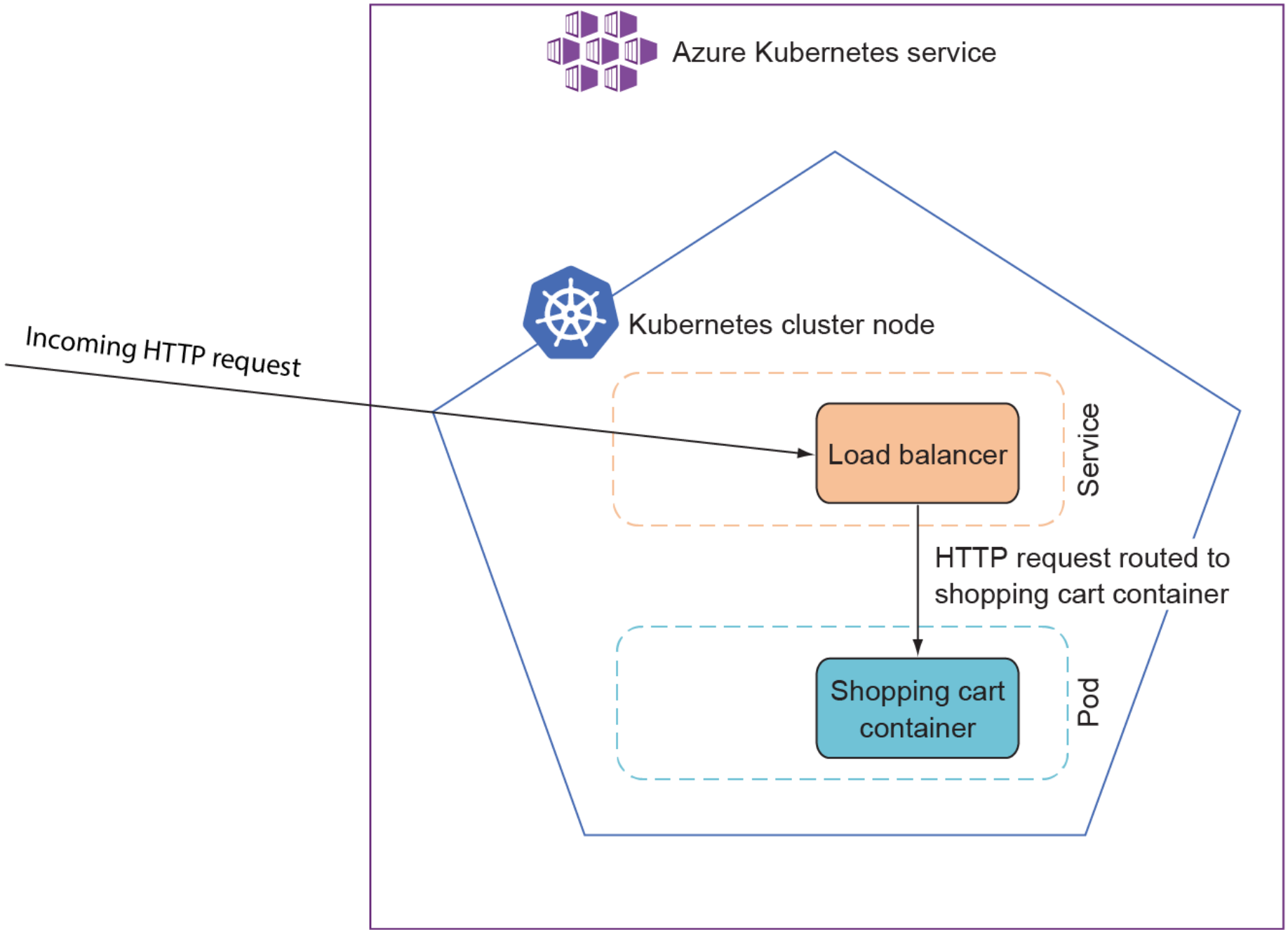
This chapter covers:

- Packaging a microservice in a Docker container
- Deploying a microservice container to Kubernetes on localhost
- Creating a basic Kubernetes cluster on Azure's AKS (Azure Kubernetes Service)
- Deploying a microservice container to a Kubernetes cluster on AKS

Choosing a production environment

Simply running our microservices on localhost isn't very interesting. We need them to run somewhere our end users can get to them and use them. There are a number of options for doing that:

- Running the microservices on your own Windows or Linux servers on-premise.
- Using a Platform as a Service (PaaS) cloud option that supports .
- Putting microservices into containers and deploying them to a cloud-specific container service like Azure's ACS or Amazon's ECS Platform as a Service (PaaS) cloud option that supports
- Using cloud-agnostic container orchestrators like Kubernetes, Apache Mesos, or RedHat OpenShift.


Azure Kubernetes service
Kubernetes cluster node
Incoming HTTP request
Load balancer
Service
HTTP request routed to
shopping cart container
Shopping cart
container
Pod

Putting the Shopping Cart microservice in a container

Adding a Dockerfile to the Shopping Cart microservice

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY ["ShoppingCart/ShoppingCart.csproj", "ShoppingCart/"]
RUN dotnet restore "ShoppingCart/ShoppingCart.csproj"
COPY . .
WORKDIR "/src/ShoppingCart"
RUN dotnet build "ShoppingCart.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "ShoppingCart.csproj" -c Release -o /app/publish


FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS final
WORKDIR /app
EXPOSE 80
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ShoppingCart.dll"]
```

Steps in the Dockerfile

The steps in the Dockerfile are as follows:

- Build the shopping cart code
- Publish the shopping cart microservice
- Create a container image based on ASP.NET

Adding a Dockerfile to the Shopping Cart microservice

- To make sure the Dockerfile runs a clean build, we add a .dockerignore with these lines that make sure any bin and obj folders are not copied into the container:

```
[B|b]in/
[O|o]bj/
```

Building and running the shopping cart container

- The next step is to build a shopping cart container image from the Dockerfile we just added.
- First, make sure you have Docker running and then open a command line and go to the root of shopping cart—where the Dockerfile is.
- Then issue this Docker command:

```
docker build . -t shopping-cart
```

Building and running the shopping cart container

- Subsequent builds will be faster.
- The output of the docker build is rather long.
- When it is successful the last few lines of the output are similar to

```
Successfully built 8d448ba53088
Successfully tagged shopping-cart:latest
```

Building and running the shopping cart container

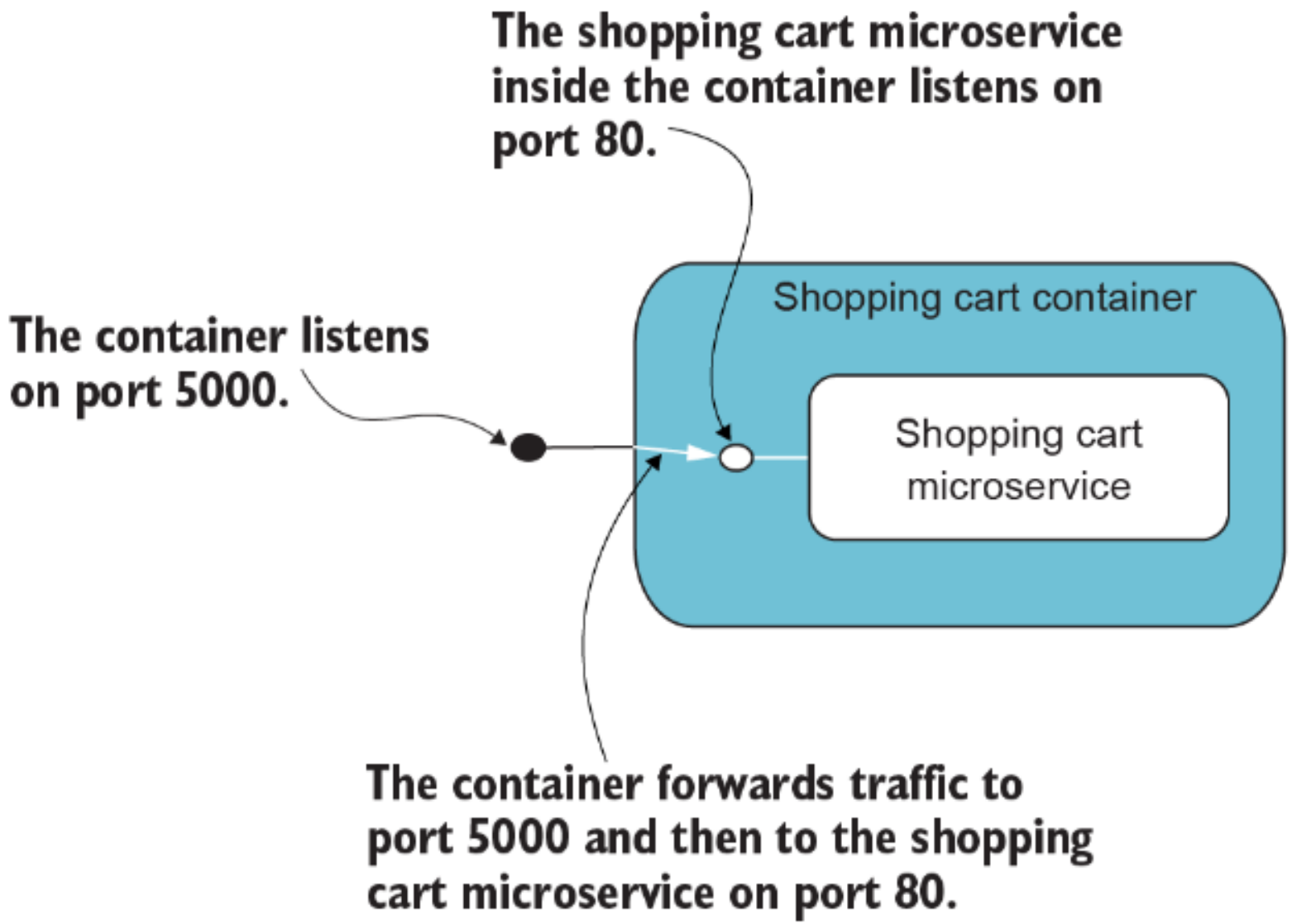
- Possibly followed by this warning if you are on Windows:

SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double-check and reset permissions for sensitive files and directories.

Building and running the shopping cart container

- You are now ready to run the newly built container image with this command:

```
> docker run --name shopping-cart --rm -p 5000:80 shopping-cart
```

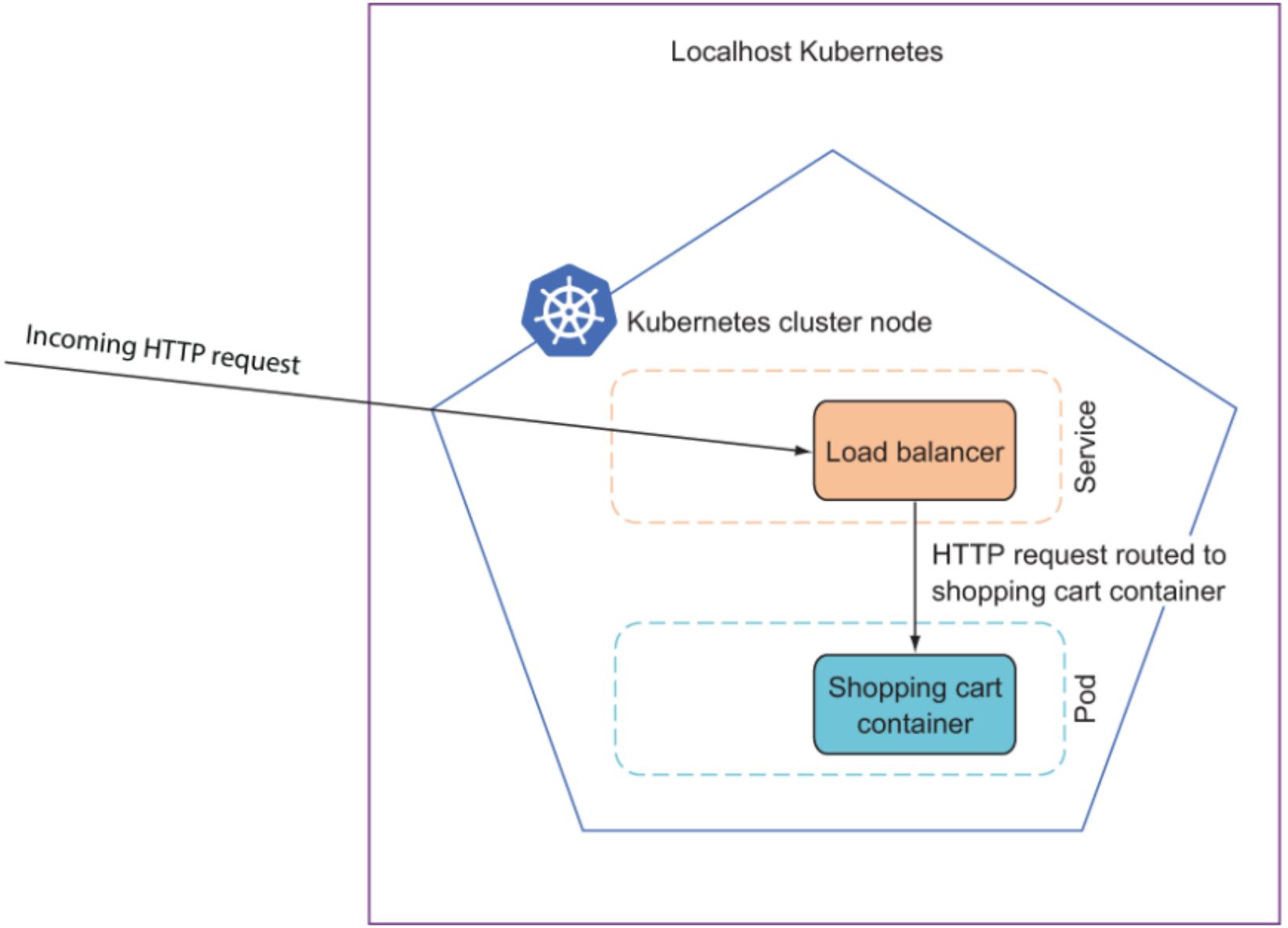

The shopping cart microservice inside the container listens on port 80.
The container listens on port 5000.
Shopping cart container
Shopping cart microservice
The container forwards traffic to port 5000 and then to the shopping cart microservice on port 80.

Building and running the shopping cart container

- To stop the shopping cart container, you can use the docker stop command:

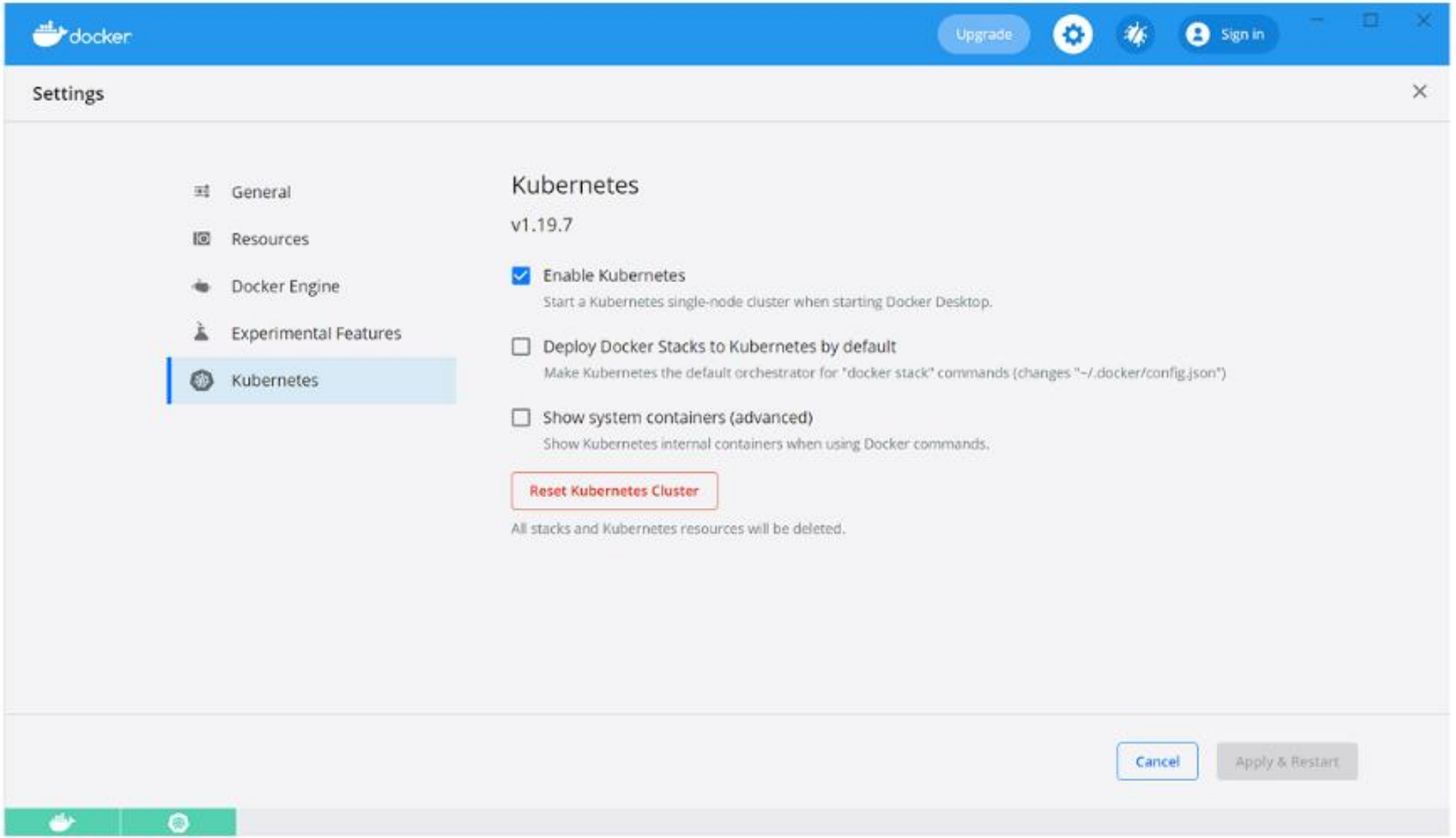
```
> docker stop shopping-cart
```

Running the shopping cart container in Kubernetes



Setting up Kubernetes localhost

Docker Desktop Kubernetes settings



- To install MicroK8S on a Linux machine, simply run this command:

sudo snap install microk8s –classic

- This will install and start the Kubernetes cluster. Furthermore, this installs the microk8s command-line interface, which includes the kubectl command we are going to use

snap alias microk8s.kubectl kubectl

- When Kubernetes is running, you can go to the command line and check that Kubernetes is indeed running, using this command

kubectl cluster-info

- Which should give you a response similar to this:

```
Kubernetes master is running at https://kubernetes.docker.internal:6443
KubeDNS is running at https://kubernetes.docker.internal:6443/api/v1/
  namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info
  dump'.
```

Creating Kubernetes deployment for the shopping cart

- Next, we want to deploy the shopping cart to the Kubernetes cluster we just installed and started. To do that, we need add a manifest file describing the to the shopping cart code base called shopping-cart.yaml. This file contains two major sections:

1. A deployment section
2. A service section

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: shopping-cart
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopping-cart
  template:
    metadata:
      labels:
        app: shopping-cart
    spec:
      containers:
        - name: shopping-cart
          image: shopping-cart
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
---
kind: Service
apiVersion: v1
metadata:
  name: shopping-cart
spec:
  type: LoadBalancer
  ports:
    - name: shopping-cart
      port: 5000
      targetPort: 80
  selector:
    app: shopping-cart
```

- Using this manifest to deploy and run the shopping cart in Kubernetes is as simple as running this from the command line:

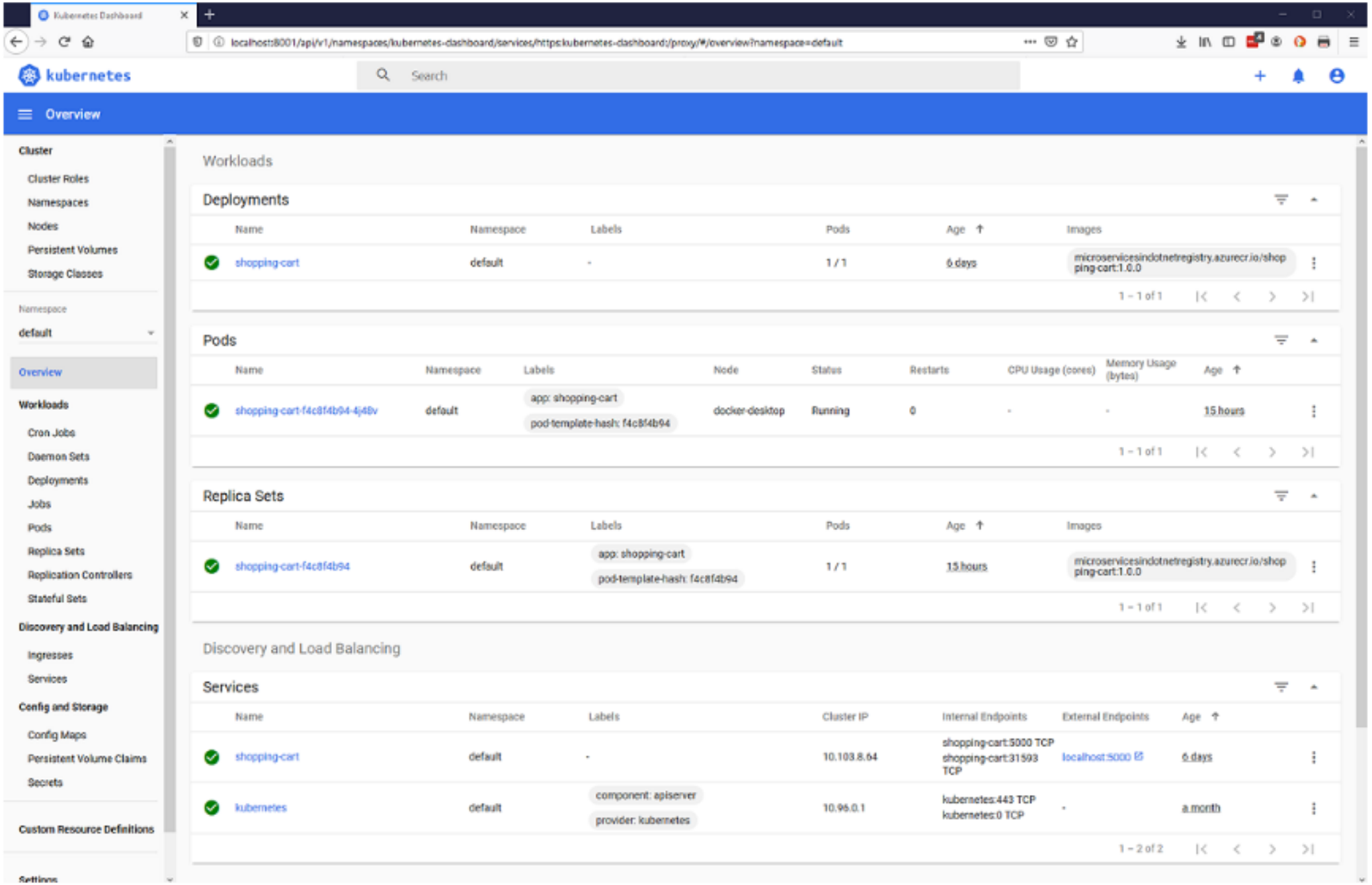
kubectl apply –f shopping-cart.yaml

- If the deployment went well, the output from kubectl get all should be similar to this:

```
NAME                                READY   STATUS    RESTARTS   AGE
pod/shopping-cart-f4c8f4b94-4j48v   1/1     Running   0          15h

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
service/kubernetes      ClusterIP      10.96.0.1     <none>        443/TCP          36d
service/shopping-cart   LoadBalancer   10.103.8.64   localhost     5000:31593/TCP   6d20h

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/shopping-cart   1/1     1            1           6d20h

NAME                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/shopping-cart-f4c8f4b94   1         1         1       15h
```

- If you prefer to have a UI, you can install and start the Kubernetes dashboard.
- First install the Kubernetes dashboard using this command:

```
> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/
 v2.2.0/aio/deploy/recommended.yaml
```


Kubernetes Dashboard
localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/overview?namespace=default
kubernetes
Search
Overview
Cluster
Cluster Roles
Namespaces
Nodes
Persistent Volumes
Storage Classes
Namespace
default
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Discovery and Load Balancing
Ingresses
Services
Config and Storage
Config Maps
Persistent Volume Claims
Secrets
Custom Resource Definitions
Settings
Workloads
Deployments
Name Namespace Labels Pods Age Images
shopping-cart default - 1 / 1 6 days microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0
1 – 1 of 1
Pods
Name Namespace Labels Node Status Restarts CPU Usage (cores) Memory Usage (bytes) Age
shopping-cart-f4c8f4b94-4j48v default app: shopping-cart pod-template-hash: f4c8f4b94 docker-desktop Running 0 - - 15 hours
1 – 1 of 1
Replica Sets
Name Namespace Labels Pods Age Images
shopping-cart-f4c8f4b94 default app: shopping-cart pod-template-hash: f4c8f4b94 1 / 1 15 hours microservicesindotnetregistry.azurecr.io/shopping-cart:1.0.0
1 – 1 of 1
Discovery and Load Balancing
Services
Name Namespace Labels Cluster IP Internal Endpoints External Endpoints Age
shopping-cart default - 10.103.8.64 shopping-cart:5000 TCP shopping-cart:31593 TCP localhost:5000 6 days
kubernetes default component: apiserver provider: kubernetes 10.96.0.1 kubernetes:443 TCP kubernetes:0 TCP - a month
1 – 2 of 2

- Let’s add a couple of items to a cart:

POST http://localhost:5000///shoppingcart/15/items

Accept: application/json

Content-Type: application/json

[1, 2]

- Next, let’s read the same cart back:

GET http://localhost:5000///shoppingcart/15

- The body of the response from the GET request should be the list of items in the cart:


```
{
  "userId": 15,
  "items": [
    {
      "productCatalogueId": 1,
      "productName": "Basic t-shirt",
      "description": "a quiet t-shirt",
      "price": {
        "currency": "eur",
        "amount": 40
      }
    },
    {
      "productCatalogueId": 2,
      "productName": "Fancy shirt",
      "description": "a loud t-shirt",
      "price": {
        "currency": "eur",
        "amount": 50
      }
    }
  ]
}
```

- Once done with the shopping cart we can clean up by removing the shopping cart from Kubernetes using this command:

```
kubectl delete -f shopping-cart.yaml
```

Running the shopping cart container on Azure Kubernetes Service

- Set up AKS. We need to create all the Azure resources for a Kubernetes cluster in AKS. This includes the cluster itself, networking, and a private container registry where we will store the container images for our microservices.

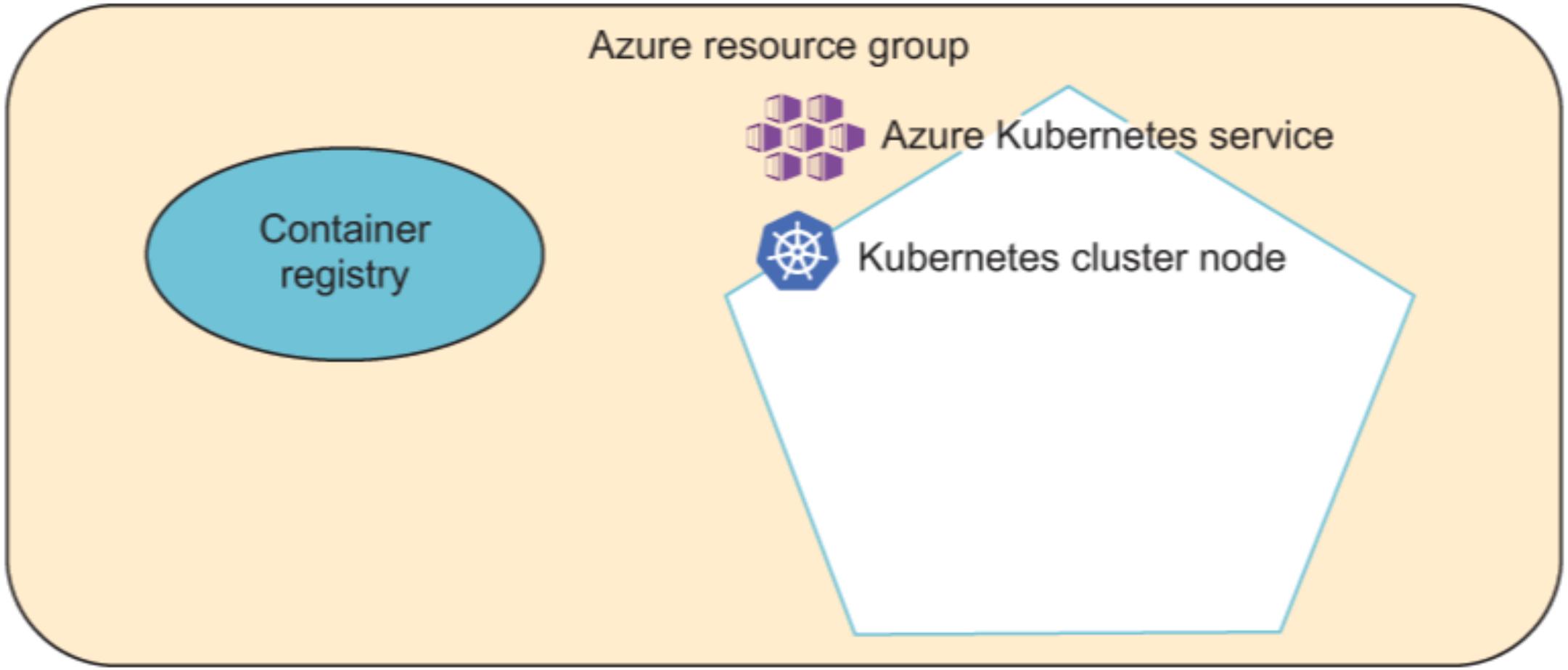
- Push the shopping cart container image to our private container registry.

- Deploy the shopping cart to the AKS cluster using the shopping cart's Kubernetes manifest.

Setting up AKS

The setup we need in order to be ready to work with a Kubernetes cluster in AKS consists of four parts:

- Creating a resource group in Azure.
- Creating a private container registry.
- Creating a Kubernetes cluster in AKS.
- Logging our local Kubernetes command line—kubectl—into the newly created AKS cluster.


Azure resource group
Azure Kubernetes service
Container registry
Kubernetes cluster node

```
az group create --name MicroservicesInDotnet --location northeurope

az acr create --resource-group MicroservicesInDotnet --name
  YOUR_UNIQUE_REGISTRY_NAME --sku Basic

az aks create --resource-group MicroservicesInDotnet --name
  MicroservicesInDotnetAKSCluster --node-count 1 --enable-addons monitoring
  --generate-ssh-keys --attach-acr YOUR_UNIQUE_REGISTRY_NAME

az aks get-credentials --resource-group MicroservicesInDotnet --name
  MicroservicesInDotnetAKSCluster
kubectl get nodes
```

- The last lines of the output are the output from the line kubectl get nodes and should look similar to this:

```
NAME                                STATUS   ROLES   AGE    VERSION
aks-nodepool1-32786309-vmss000000   Ready    agent   107s   v1.14.8
```

- When you are done with the Kubernetes cluster in AKS, you can delete the cluster as well as the container registry with this command:

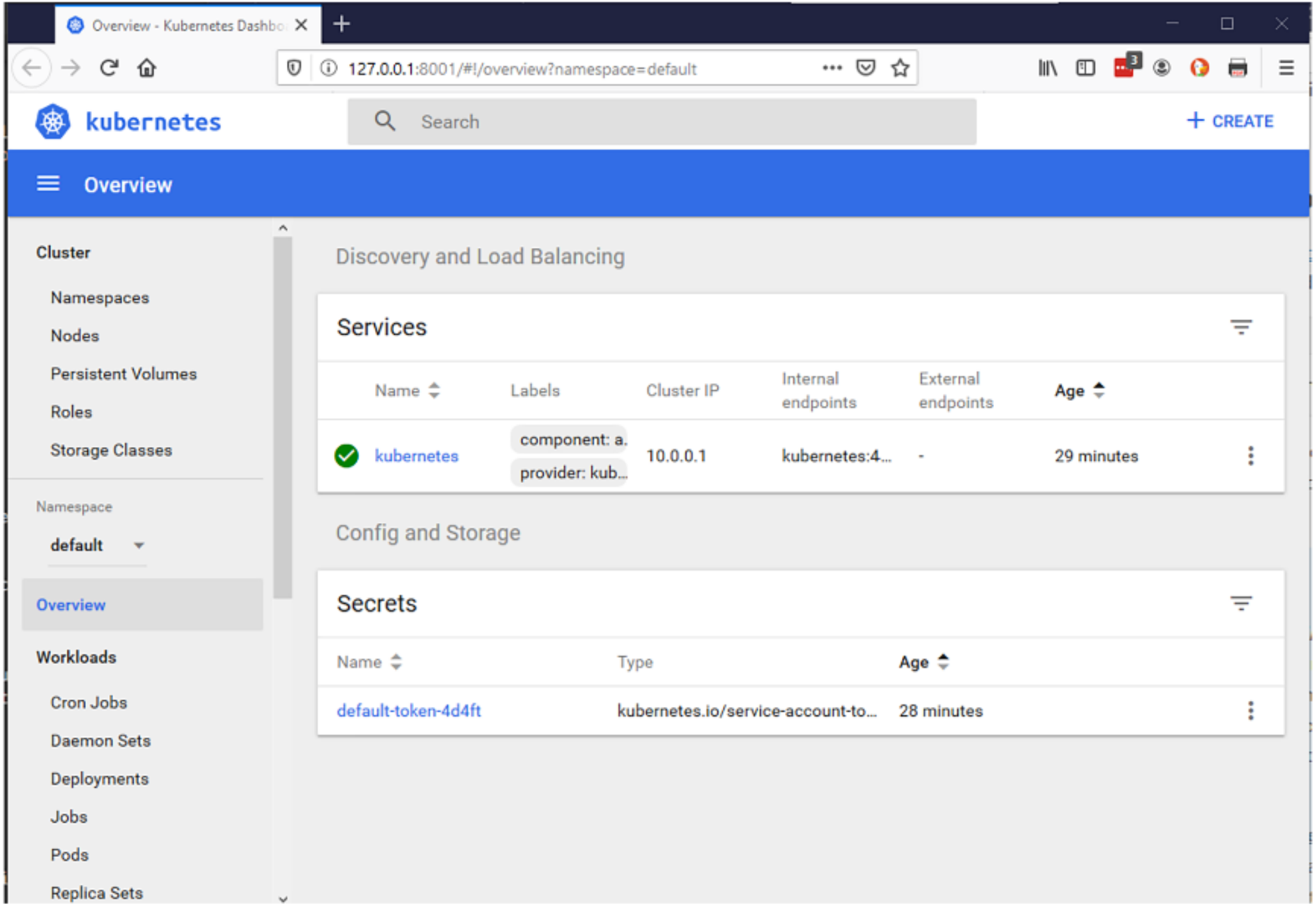
```
> az group delete --name MicroservicesInDotnet --yes --no-wait
```

- This command will give the dashboard access by telling Kubernetes to assign the role cluster-admin to the account kube-system:kubernetes-dashboard:

```
> kubectl create clusterrolebinding kubernetes-dashboard
  --clusterrole=cluster-admin
  --serviceaccount=kube-system:kubernetes-dashboard
```

- With the access in place we can start the Kubernetes dashboard with the Azure command line like this:

```
>az aks browse --resource-group MicroservicesInDotnet  --name
MicroservicesInDotnetAKSCluster
```


Overview - Kubernetes Dashbo
127.0.0.1:8001/#!/overview?namespace=default
kubernetes
Search
+ CREATE
Overview
Cluster
Namespaces
Nodes
Persistent Volumes
Roles
Storage Classes
Namespace
default
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Discovery and Load Balancing
Services
Name
Labels
Cluster IP
Internal endpoints
External endpoints
Age
kubernetes
component: a.
provider: kub...
10.0.0.1
kubernetes:4...
-
29 minutes
Config and Storage
Secrets
Name
Type
Age
default-token-4d4ft
kubernetes.io/service-account-to...
28 minutes

Running the shopping cart in AKS

The remaining steps to deploying and running the shopping cart in AKS are as follows:

- Tag the shopping cart container image, so we have a precise identification of the container image.
- Push the tagged container image to our container registry in Azure.
- Modify the shopping cart's Kubernetes manifest to refer to the tagged container.
- Apply the modified manifest to AKS.
- Test that the shopping cart runs correctly.

- We set the tag as follows:

```
>docker tag shopping-cart your_unique_registry_name.azurecr.io/shopping-cart:1.0.0
```

- This authentication is done using the Azure CLI like this:

```
> az acr login --name YOUR_UNIQUE_REGISTRY_NAME
```

- Now the tagged container image can be pushed to our private Docker registry in Azure with this Docker command:

```
> docker push your_unique_registry_name.azurecr.io/shopping-cart:1.0.0
```

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: shopping-cart
spec:
  replicas: 1
  selector:
    matchLabels:
      app: shopping-cart
  template:
    metadata:
      labels:
        app: shopping-cart
    spec:
      containers:
        - name: shopping-cart
          image: your_unique_registry_name.azurecr.io/
            shopping-cart:1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: shopping-cart
spec:
  type: LoadBalancer
  ports:
    - name: shopping-cart
      port: 5000
      targetPort: 80
  selector:
    app: shopping-cart
```

- The only change is the image name, which now refers to the image in our private container registry in Azure.

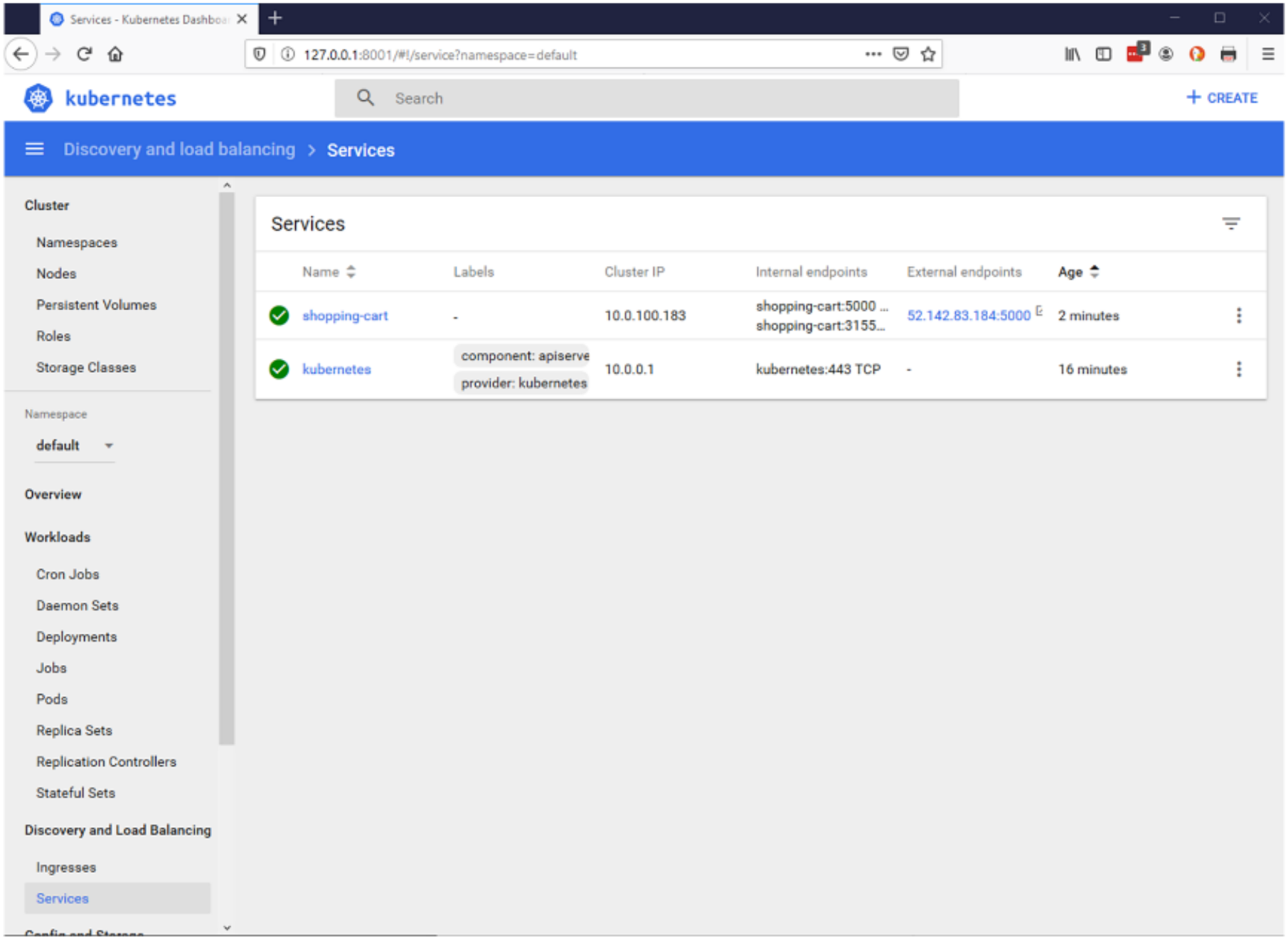
- All that remains is to apply this manifest to the Kubernetes cluster in AKS:

```
> kubectl apply -f shopping-cart.yaml
```

- Using the command line the command kubectl get all will show the information about everything running in the cluster.

- The output should be similar to this:

```
NAME                                 READY   STATUS    RESTARTS   AGE
pod/shopping-cart-f4c8f4b94-vnmn9    1/1     Running   0          107s

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
service/kubernetes      ClusterIP      10.0.0.1      <none>         443/TCP          15m
service/shopping-cart   LoadBalancer   10.0.100.183  52.142.83.184  5000:31552/TCP   107s

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/shopping-cart     1/1     1            1           107s

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/shopping-cart-f4c8f4b94      1         1         1       107s
```


Services - Kubernetes Dashboard
127.0.0.1:8001/#!/service?namespace=default
kubernetes
Search
+ CREATE
Discovery and load balancing > Services
Cluster
Namespaces
Nodes
Persistent Volumes
Roles
Storage Classes
Namespace
default
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Discovery and Load Balancing
Ingresses
Services
Services
Name
Labels
Cluster IP
Internal endpoints
External endpoints
Age
shopping-cart
-
10.0.100.183
shopping-cart:5000 ...
shopping-cart:3155...
52.142.83.184:5000
2 minutes
kubernetes
component: apiserve
provider: kubernetes
10.0.0.1
kubernetes:443 TCP
-
16 minutes

- To verify that the shopping cart is indeed running, we can test its endpoint by, for instance, adding items to a cart using the shopping cart's POST endpoint

POST http:52.142.83.184:5000///shoppingcart/15/items

Accept: application/json

Content-Type: application/json

[1, 2]

- And then reading the same cart using the GET endpoint:

GET http:52.142.83.184:5000///shoppingcart/15

Summary

- The microservices we develop can be deployed to many different environments.
- .NET-based microservices are easily put into containers and run as containers.
- Dockerfiles for .NET-based microservices follow a common template.
- Deploying our microservices to Kubernetes gives us a highly scalable environment and a versatile container orchestrator.
- Kubernetes works the same on localhost and in the cloud. This means we can easily run the exact containers on localhost for development and in the cloud for production.
- Kubernetes can run everything we are going to develop in the upcoming chapters while providing tools for scaling, monitoring, and debugging microservices.
- Azure Kubernetes Services is an easy-to-set-up managed Kubernetes offering that enables us to get up and running with Kubernetes quickly.

Identifying and scoping microservices

Introduction

This chapter covers:

- Scoping microservices for business capability
- Scoping microservices to support technical capabilities
- Scoping microservices to support efficient development work
- Managing when scoping microservices is difficult
- Carving out new microservices from existing ones

The primary driver for scoping microservices: Business capabilities

What is a business capability?

- A *business capability* is something an organization does that contributes to business goals.

- For instance, handling a shopping cart on an e-commerce website is a business capability that contributes to the broader business goal of allowing users to purchase items.

- A given business will have a number of business capabilities that together make the overall business function.

Identifying business capabilities

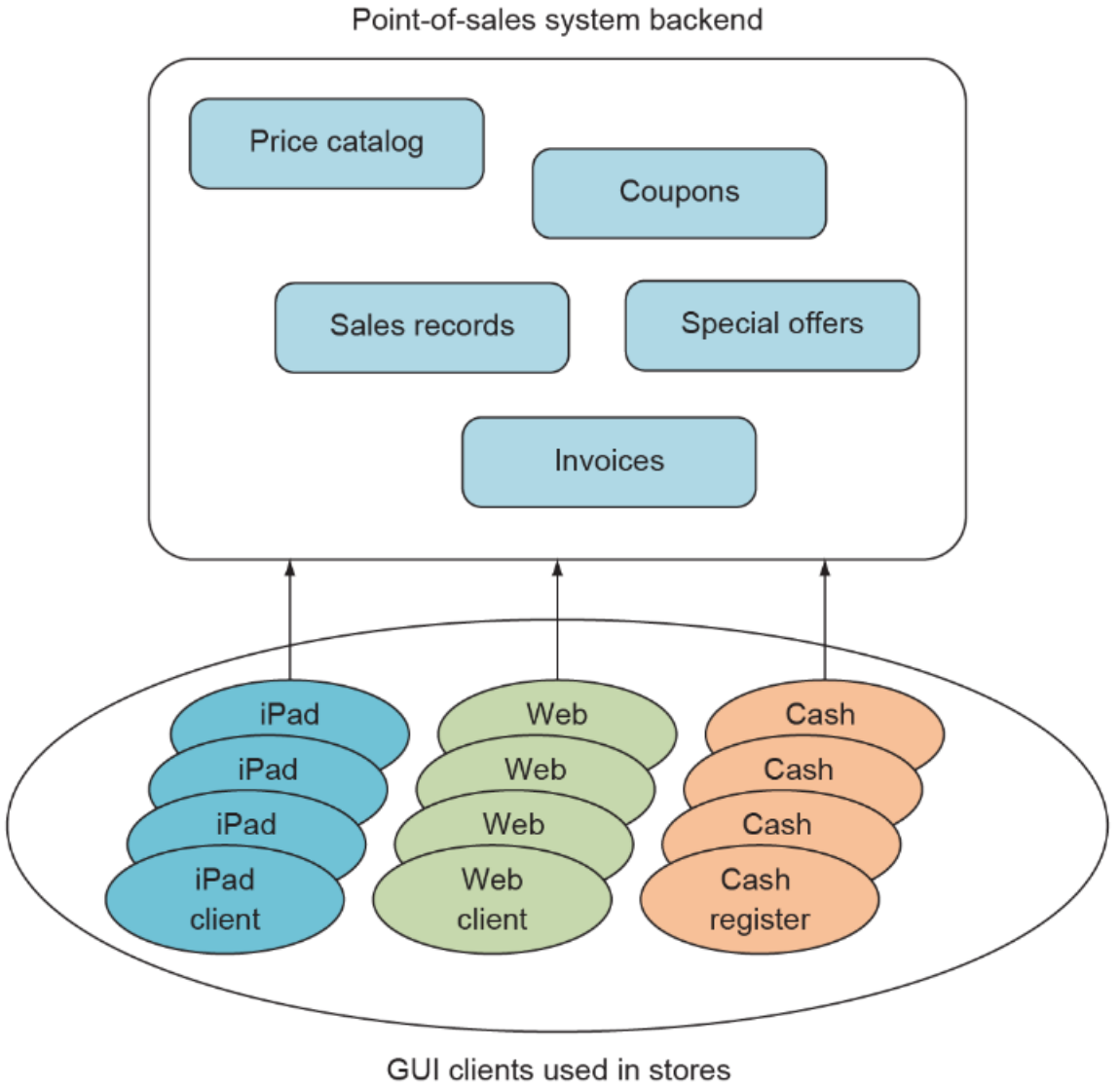
- A good understanding of the domain will enable you to understand how the business functions.

- Understanding how the business functions means you can identify the business capabilities that make up the business and the processes involved in delivering the capabilities.

- In other words, the way to identify business capabilities is to learn about the business's domain.

- You can gain this type of knowledge by talking with the people who know the business domain best: business analysts, the end users of your software, and so on—all the people directly involved in the day-to-day work that drives the business.

Example: Point-of-sale system

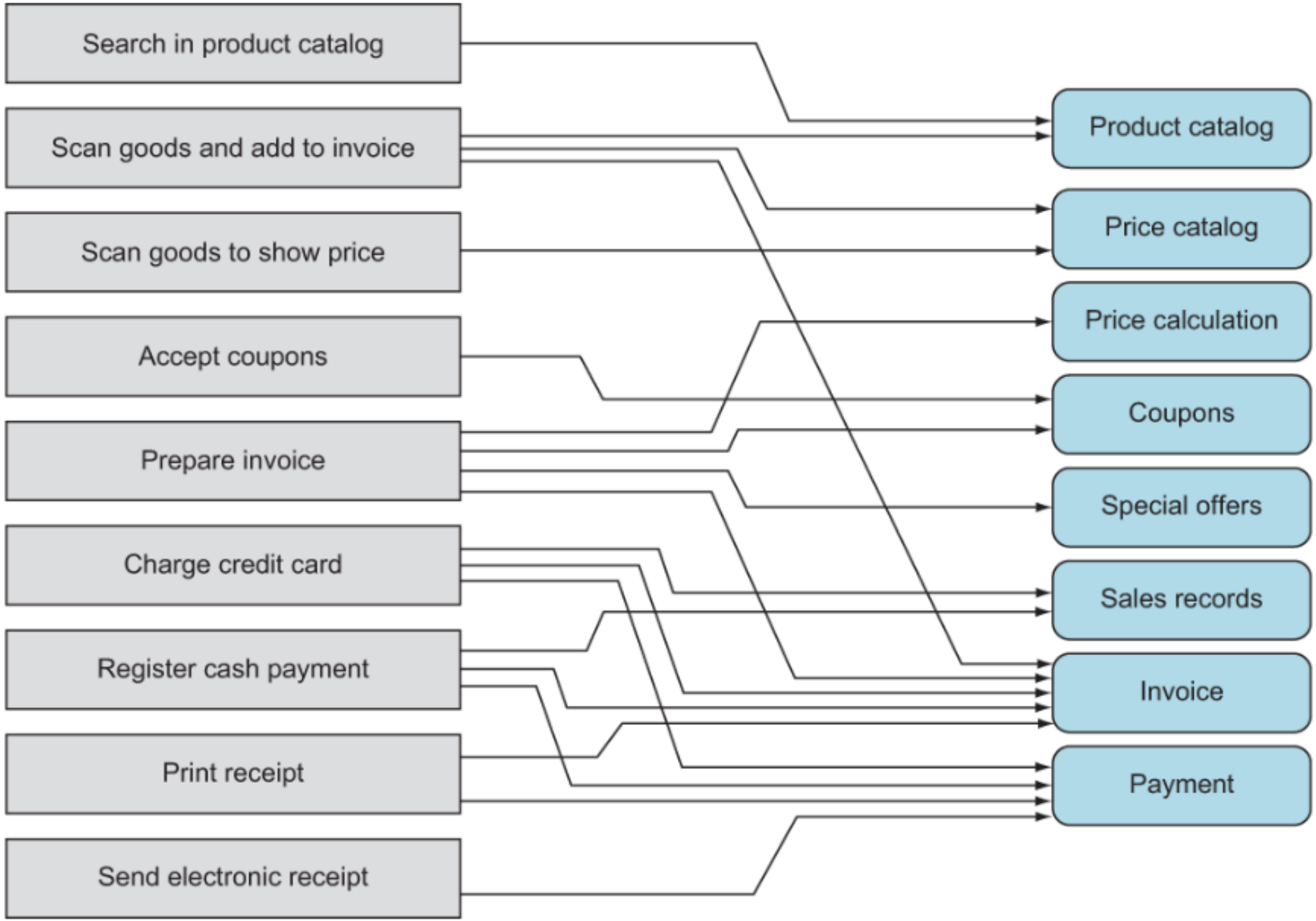


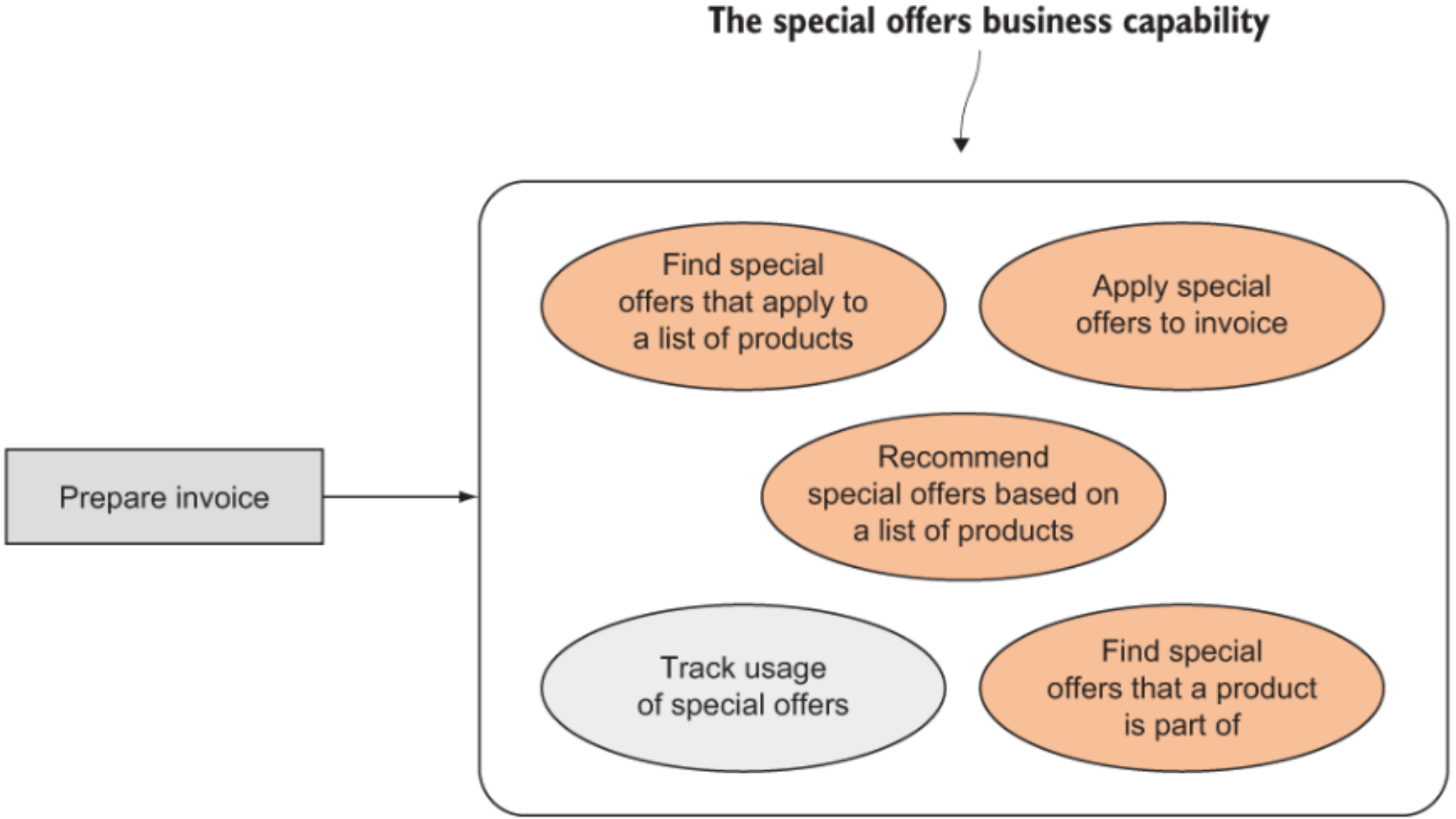
The system offers cashiers a variety of functions:

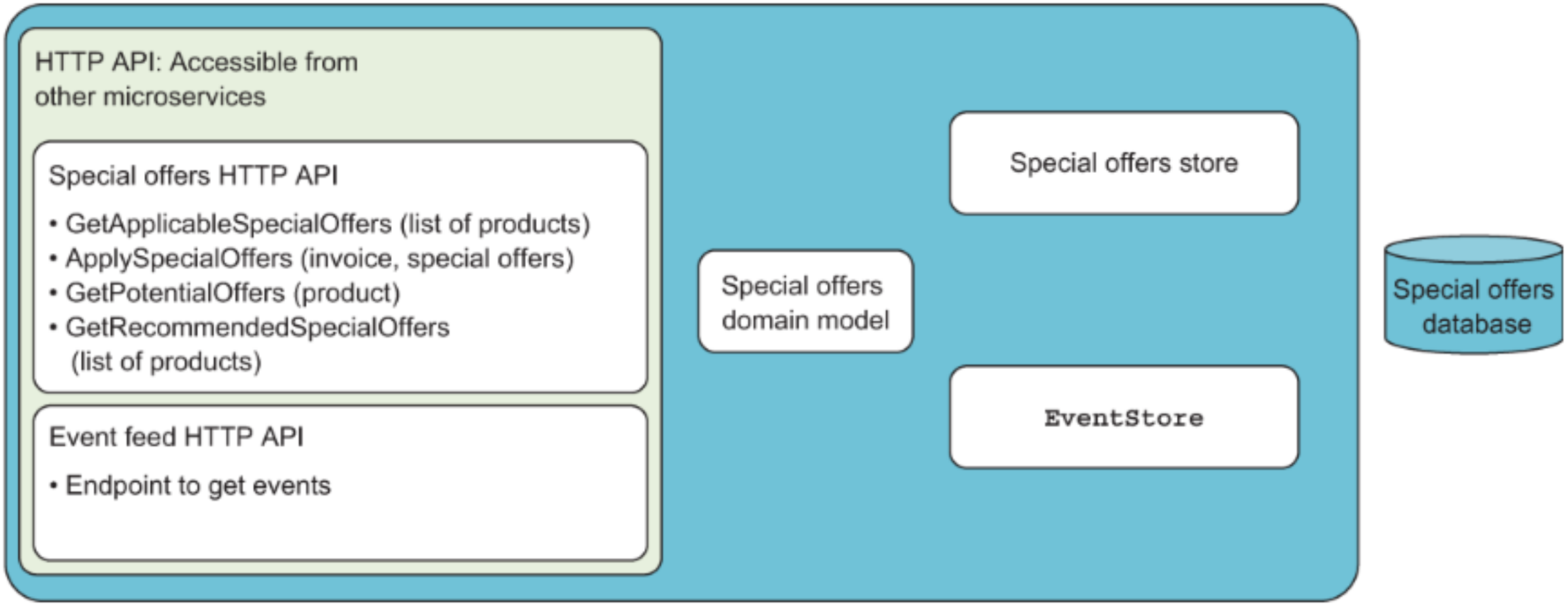
- Scan products and add them to the invoice.
- Prepare an invoice.
- Charge a credit card via a card reader attached to the client.
- Register a cash payment.
- Accept coupons.
- Print a receipt.
- Send an electronic receipt to the customer.
- Search in the product catalog.
- Scan one or more products to show prices and special offers related to the products.

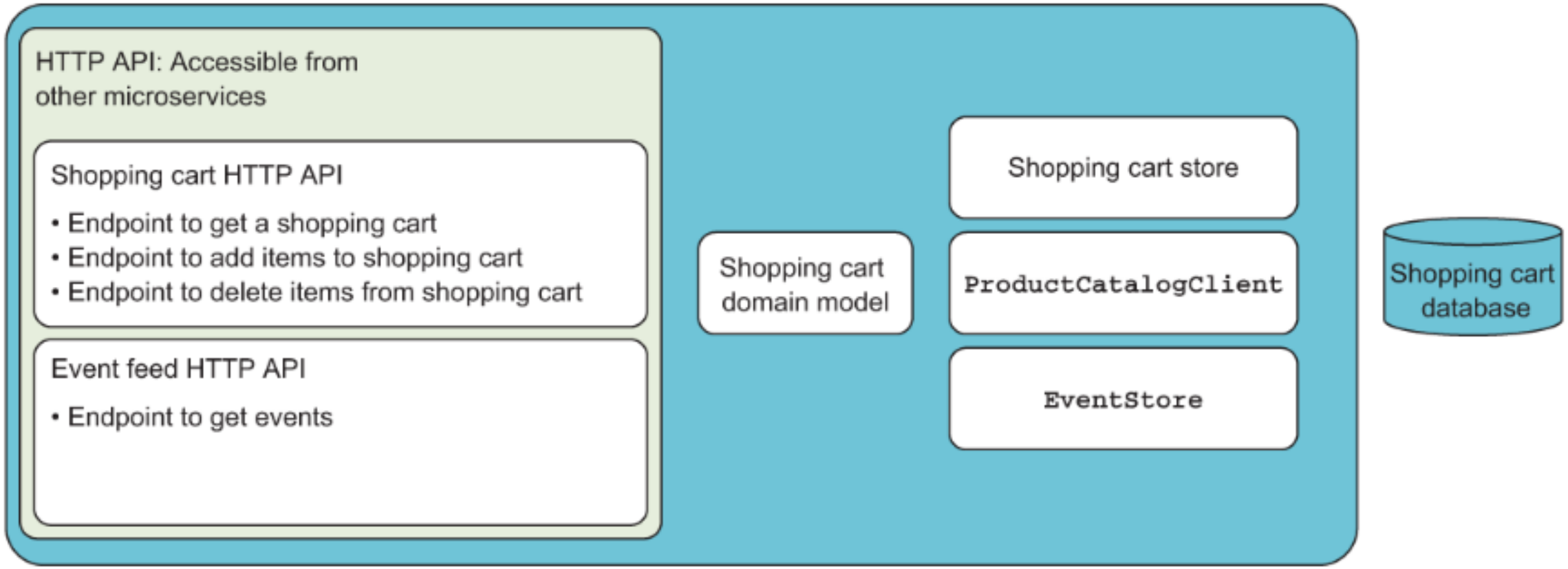
If you continued the hunt for business capabilities in the POS system, you might end up with this list:

- Product catalog
- Price catalog
- Price calculation
- Special offers
- Coupons
- Sales records
- Invoice
- Payment


Search in product catalog
Scan goods and add to invoice
Scan goods to show price
Accept coupons
Prepare invoice
Charge credit card
Register cash payment
Print receipt
Send electronic receipt
Product catalog
Price catalog
Price calculation
Coupons
Special offers
Sales records
Invoice
Payment


The special offers business capability
Prepare invoice
Find special offers that apply to a list of products
Apply special offers to invoice
Recommend special offers based on a list of products
Track usage of special offers
Find special offers that a product is part of


HTTP API: Accessible from other microservices
Special offers HTTP API
• GetApplicableSpecialOffers (list of products)
• ApplySpecialOffers (invoice, special offers)
• GetPotentialOffers (product)
• GetRecommendedSpecialOffers (list of products)
Event feed HTTP API
• Endpoint to get events
Special offers domain model
Special offers store
EventStore
Special offers database


HTTP API: Accessible from other microservices
Shopping cart HTTP API
• Endpoint to get a shopping cart
• Endpoint to add items to shopping cart
• Endpoint to delete items from shopping cart
Event feed HTTP API
• Endpoint to get events
Shopping cart domain model
Shopping cart store
ProductCatalogClient
EventStore
Shopping cart database

The secondary driver for scoping microservices: Supporting technical capabilities

- The secondary way to identify scopes for microservices is to look at supporting technical capabilities.

- A *supporting technical capability* is something that doesn't directly contribute to a business goal but supports other microservices, such as integrating with another system or scheduling an event to happen some time in the future.

What is a technical capability?

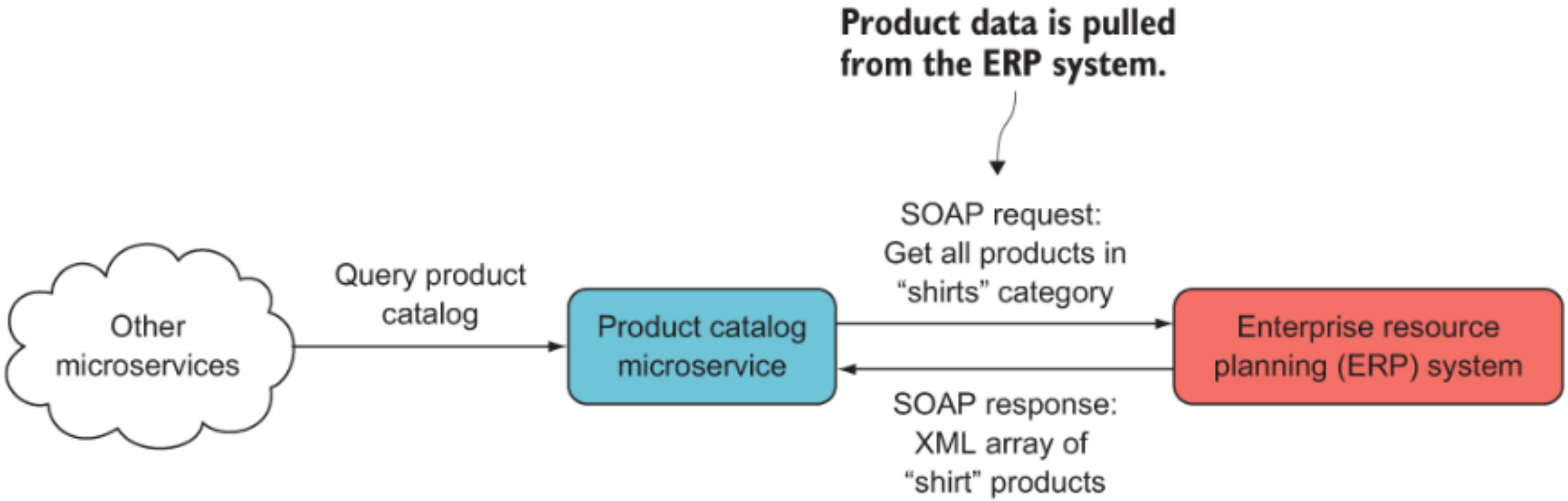
- Supporting technical capabilities are secondary drivers in scoping microservices because they don’t directly contribute to the system’s business goals.

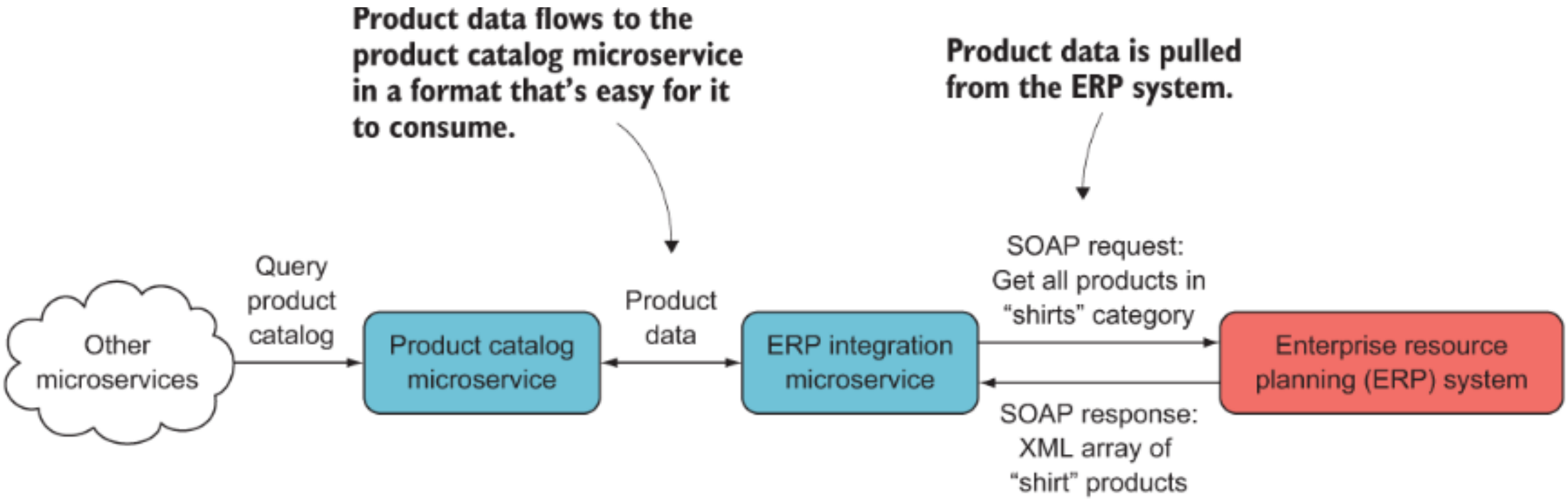
- They exist to simplify and support the other microservices that implement business capabilities.

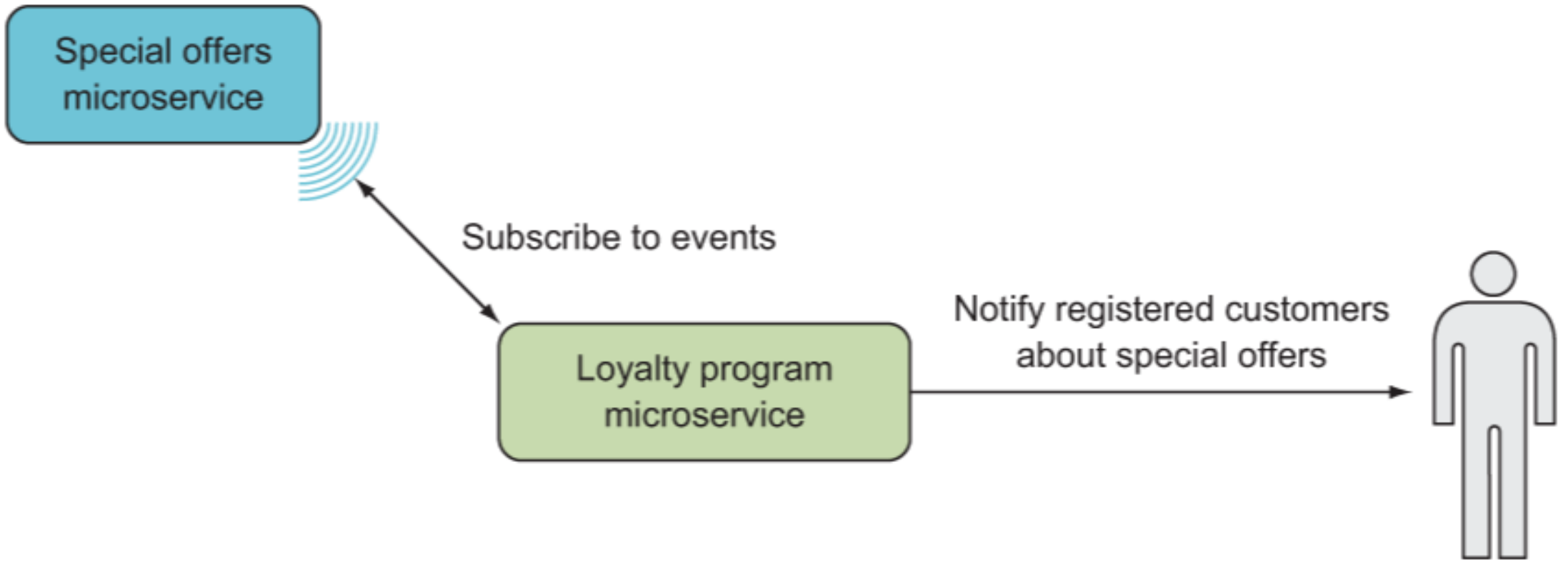
Examples of supporting technical capabilities

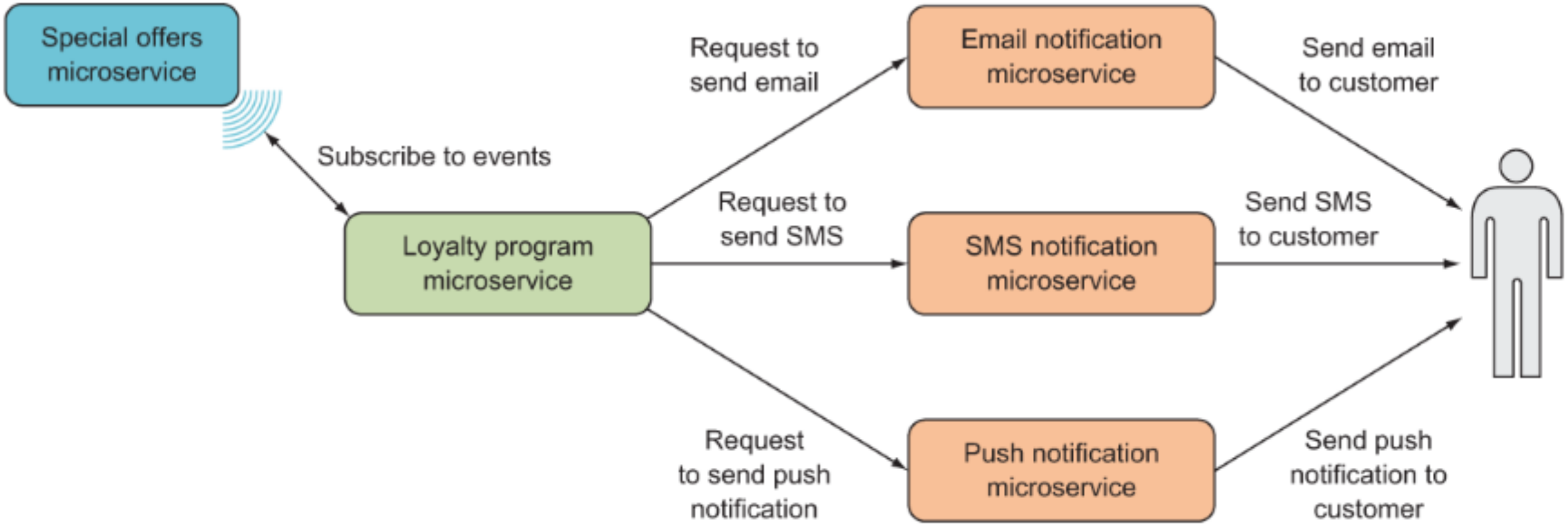
To give you a feel for what I mean by supporting technical capabilities, let's consider two examples:

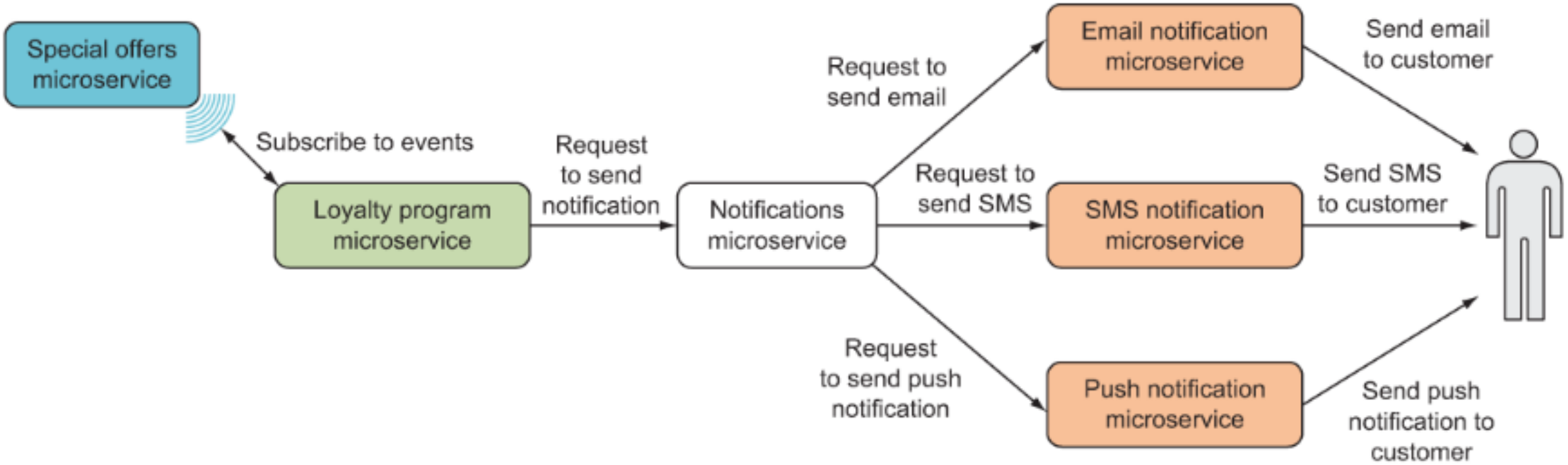
- An integration with another system.
- The ability to send notifications to customers.

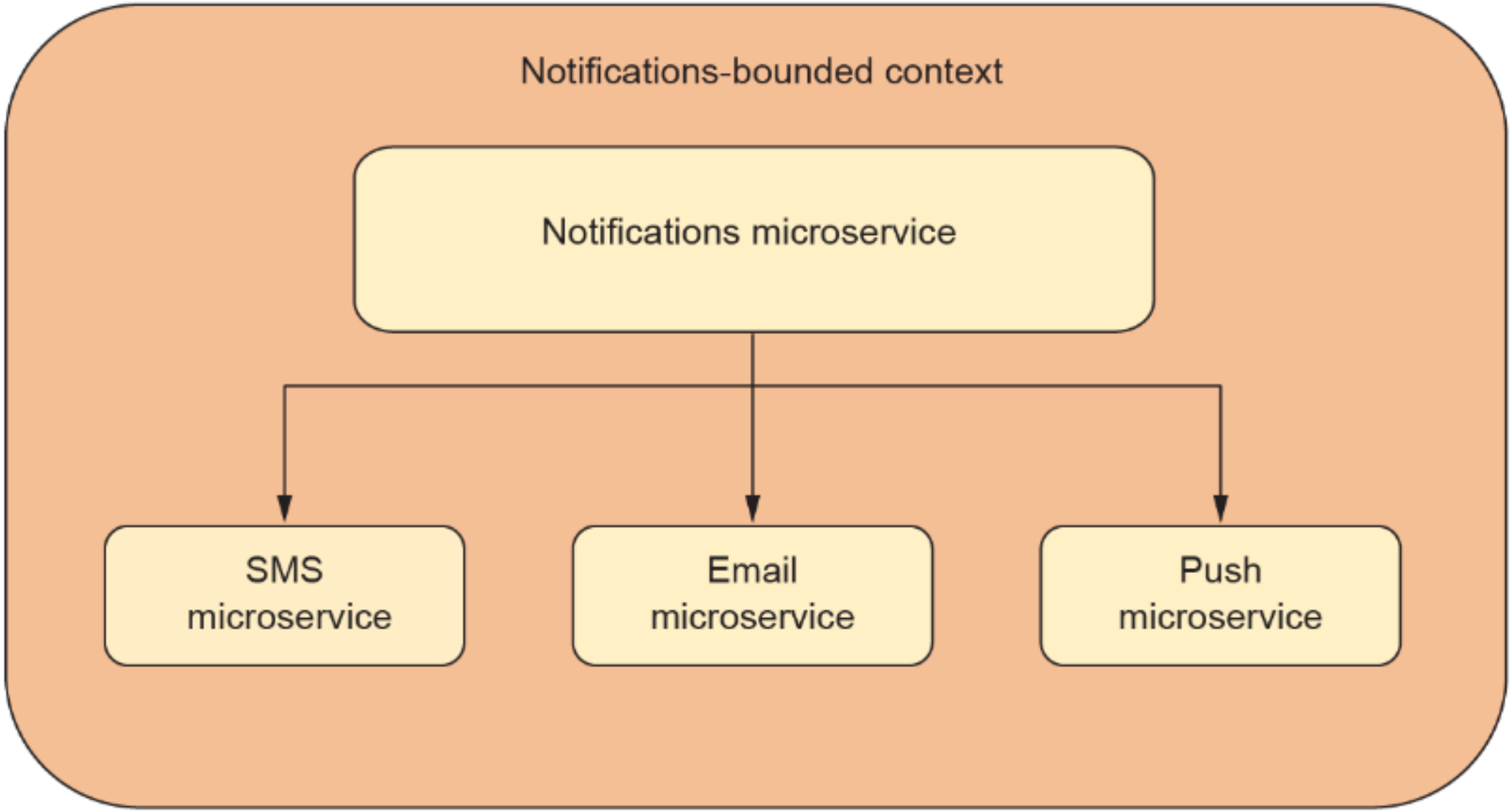

Product data is pulled from the ERP system.
SOAP request: Get all products in "shirts" category
Other microservices
Query product catalog
Product catalog microservice
Enterprise resource planning (ERP) system
SOAP response: XML array of "shirt" products


Product data flows to the product catalog microservice in a format that's easy for it to consume.
Product data is pulled from the ERP system.
Other microservices
Query product catalog
Product catalog microservice
Product data
ERP integration microservice
SOAP request: Get all products in "shirts" category
Enterprise resource planning (ERP) system
SOAP response: XML array of "shirt" products


Special offers microservice
Subscribe to events
Loyalty program microservice
Notify registered customers about special offers


Special offers microservice
Subscribe to events
Loyalty program microservice
Request to send email
Email notification microservice
Send email to customer
Request to send SMS
SMS notification microservice
Send SMS to customer
Request to send push notification
Push notification microservice
Send push notification to customer


Special offers microservice
Subscribe to events
Loyalty program microservice
Request to send notification
Notifications microservice
Request to send email
Email notification microservice
Send email to customer
Request to send SMS
SMS notification microservice
Send SMS to customer
Request to send push notification
Push notification microservice
Send push notification to customer


Notifications-bounded context
Notifications microservice
SMS microservice
Email microservice
Push microservice

Identifying technical capabilities

The question of whether a supporting technical capability should be implemented in its own microservice is about what will be easiest in the long run. You should ask these questions:

- If the supporting technical capability stays in a microservice scoped to a business capability, is there a risk that the microservice will no longer be replaceable with reasonable effort?
- Is the supporting technical capability implemented in several microservices scoped to business capabilities?
- Will a microservice implementing the supporting capability be individually deployable?
- Will all microservices scoped to business capabilities still be individually deployable if the supporting technical capability is implemented in a separate microservice?

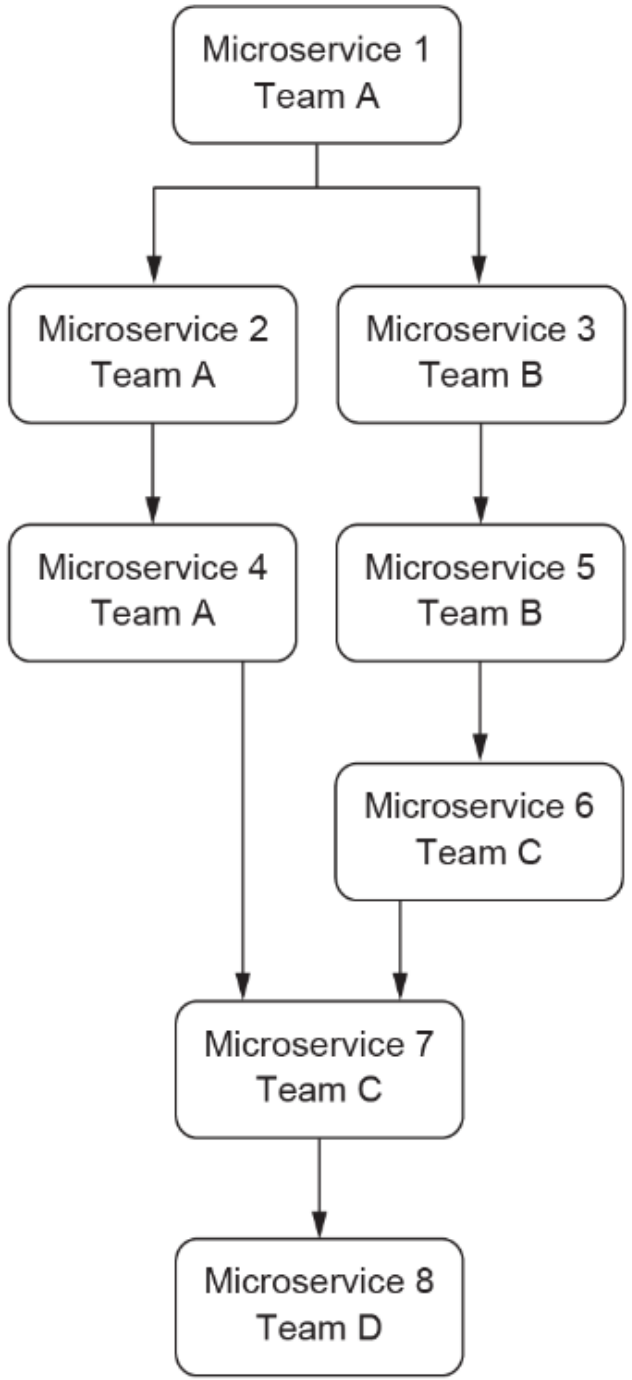
The tertiary driver for scoping microservices: Supporting efficiency of work

- The third driver when we find the scope and boundaries of microservices is supporting efficiency of work.

- The microservices we design and create should be efficient to work with and should let teams that develop and maintain the microservices work efficiently.

- This, largely, is a matter of respecting and using Conway's law, which you may recall from earlier:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

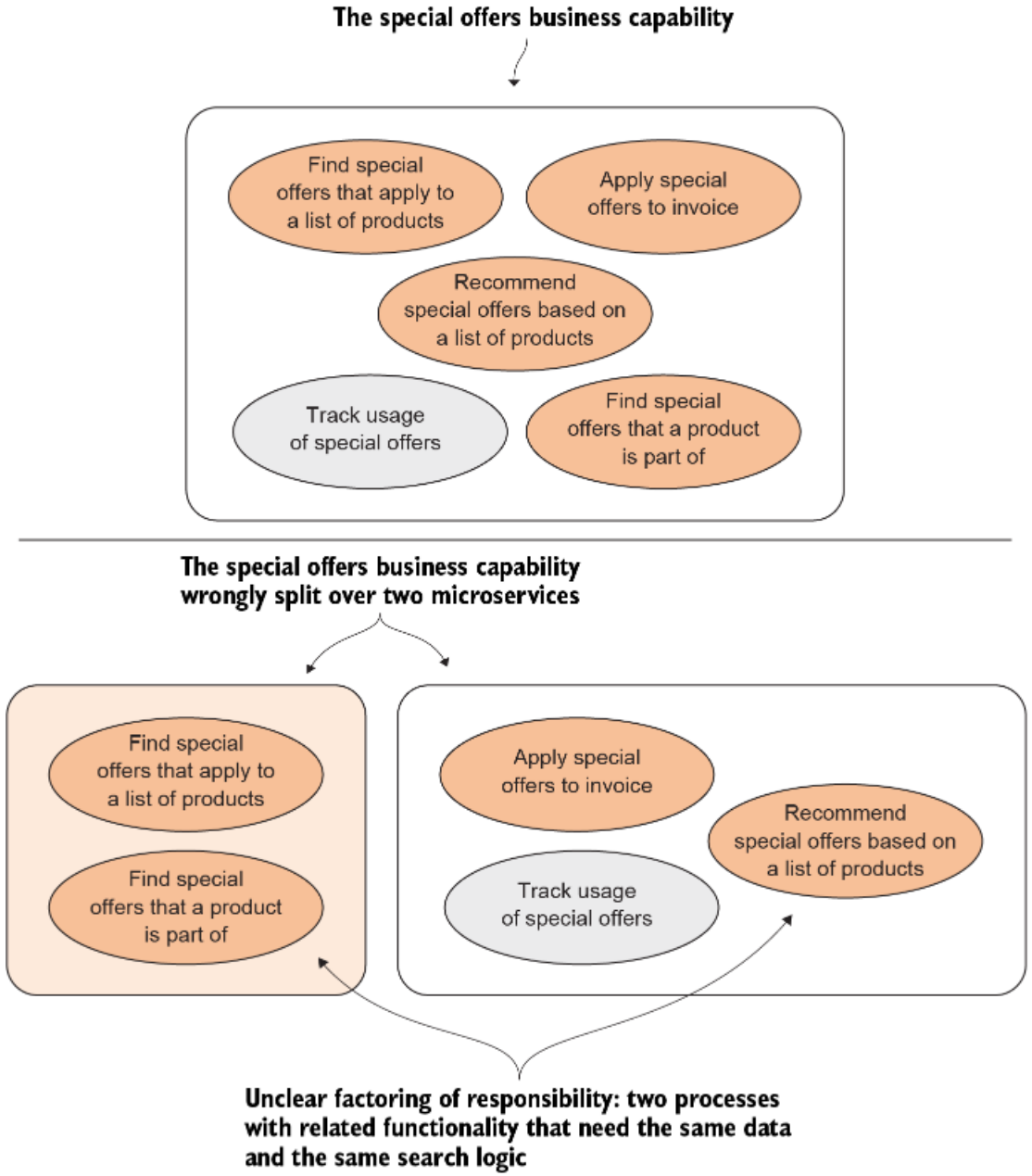

Microservice 1
Team A
Microservice 2
Team A
Microservice 3
Team B
Microservice 4
Team A
Microservice 5
Team B
Microservice 6
Team C
Microservice 7
Team C
Microservice 8
Team D

What to do when the correct scope isn’t clear

Lack of clarity can have several causes, including the following:

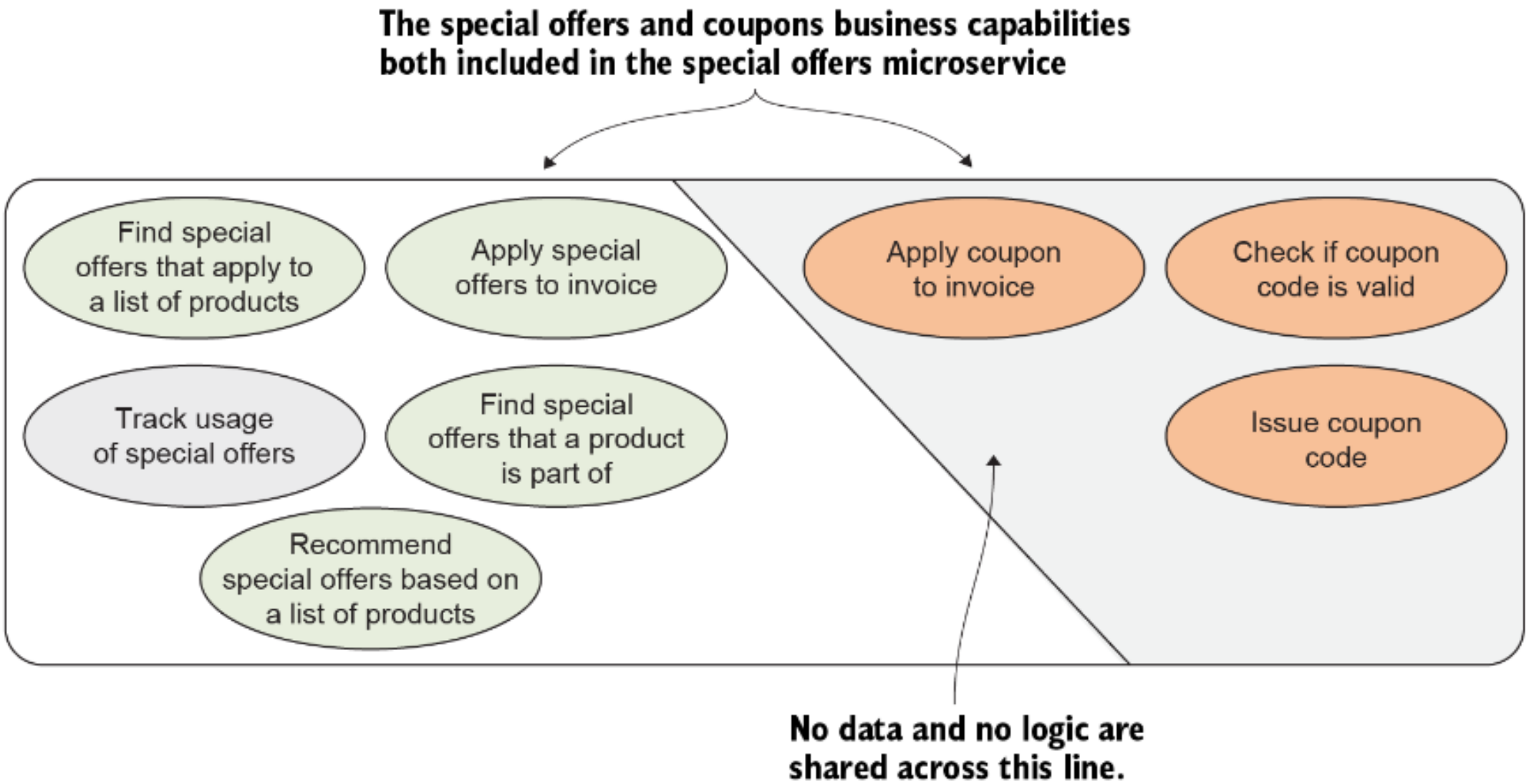
- Insufficient understanding of the business domain
- Confusion in the business domain
- Incomplete knowledge of the details of a technical capability
- Inability to estimate the complexity of a technical capability

Starting a bit bigger



If you base the scope of your microservices on only part of the special offers business capability, you'll incur some significant costs:

- Data and data model duplication between the two microservices
- Unclear factoring of responsibility
- Obstacles to refactoring the code for the business capability
- Difficulty deploying the two microservices independently

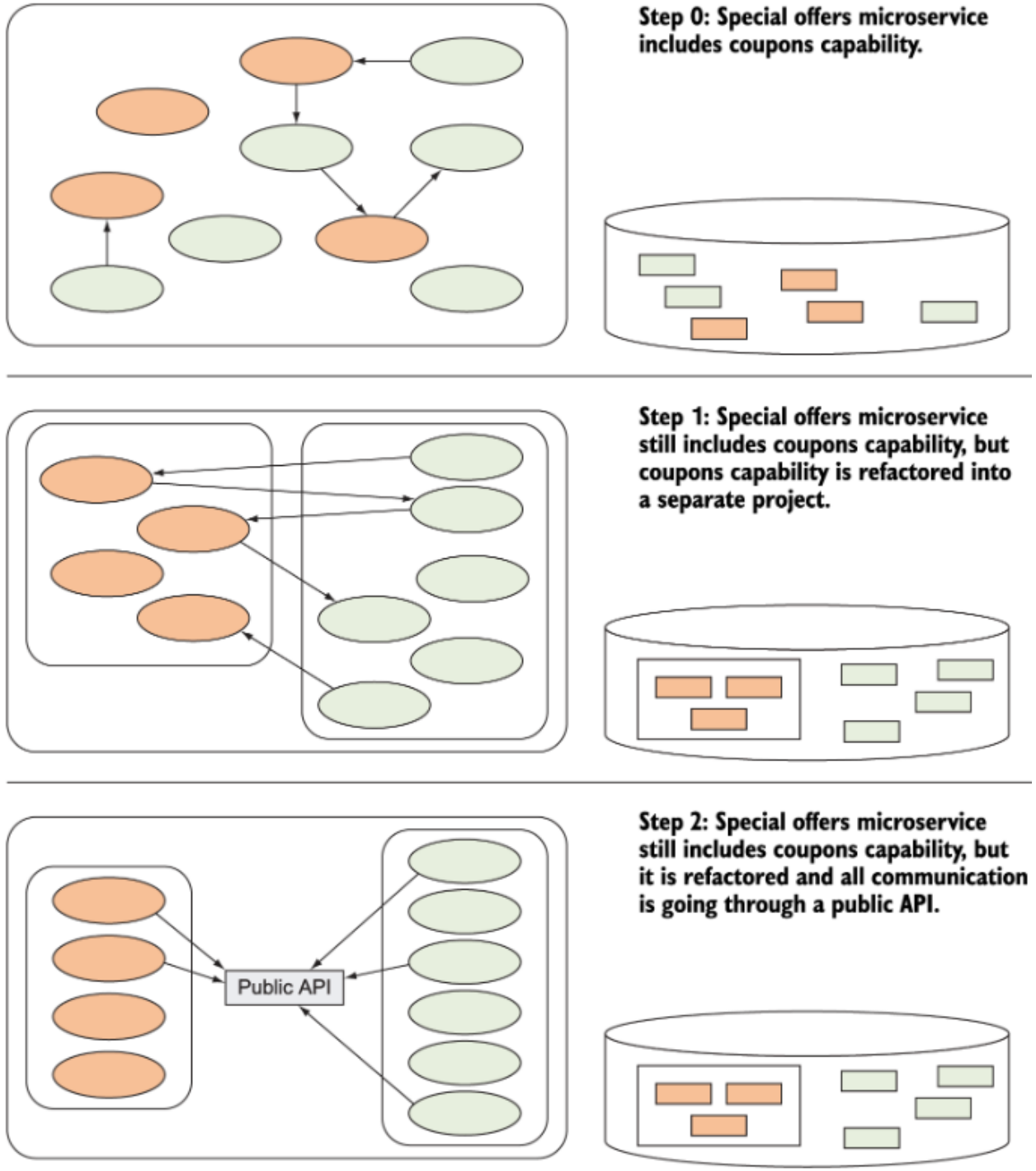

The special offers and coupons business capabilities both included in the special offers microservice
Find special offers that apply to a list of products
Apply special offers to invoice
Apply coupon to invoice
Check if coupon code is valid
Track usage of special offers
Find special offers that a product is part of
Issue coupon code
Recommend special offers based on a list of products
No data and no logic are shared across this line.

There are costs associated with including too much in the scope of a microservice:

- The code base becomes bigger and more complex, which can lead to changes being more expensive.

- The microservice is harder to replace.

Carving out new microservices from existing microservices



Well-scoped microservices adhere to the microservice characteristics

Characteristics of microservices:

- A microservice is responsible for a single capability.
- A microservice is individually deployable.
- A small team can maintain a handful of microservices.
- A microservice is replaceable.

Primary scoping to business capabilities leads to good microservices

The primary driver for scoping microservices is identifying business capabilities. Let’s see how that makes for microservices that adhere to the microservice characteristics.

- Responsible for a single capability
- Individually deployable
- Replaceable and maintainable by a small team

Secondary scoping to support technical capabilities leads to good microservices

The secondary driver for scoping microservices is identifying supporting technical capabilities. Let's see how that makes for microservices that adhere to the microservice characteristics.

- Responsible for a single capability
- Individually deployable
- Replaceable and maintainable by a small team

Tertiary scoping to support efficiency of work

The third driver for scoping microservices is supporting the efficiency of work. Let’s see how making sure teams can work efficiently aligns with the microservice characteristics.

- Individually deployable
- A small team can maintain a few handfuls of microservices

Summary

- Identifying supporting technical capabilities is an opportunistic form of design. You should only pull a supporting technical capability into a separate microservice if it will be an overall simplification.

- The tertiary driver in scoping microservices is efficiency of work. A team assigned to develop and maintain microservices should be able to work efficiently and autonomously.

- When you're in doubt about the scope of a microservice, lean toward making the scope slightly bigger rather than slightly smaller.

- Because scoping microservices well is difficult, you'll probably be in doubt sometimes. You're also likely to get some of the scopes wrong in your first iteration.

- You must expect to have to carve new microservices out of existing ones from time to time.

- You can use your doubt about scope to organize the code in your microservices so that they lend themselves to carving out new microservices at a later stage.