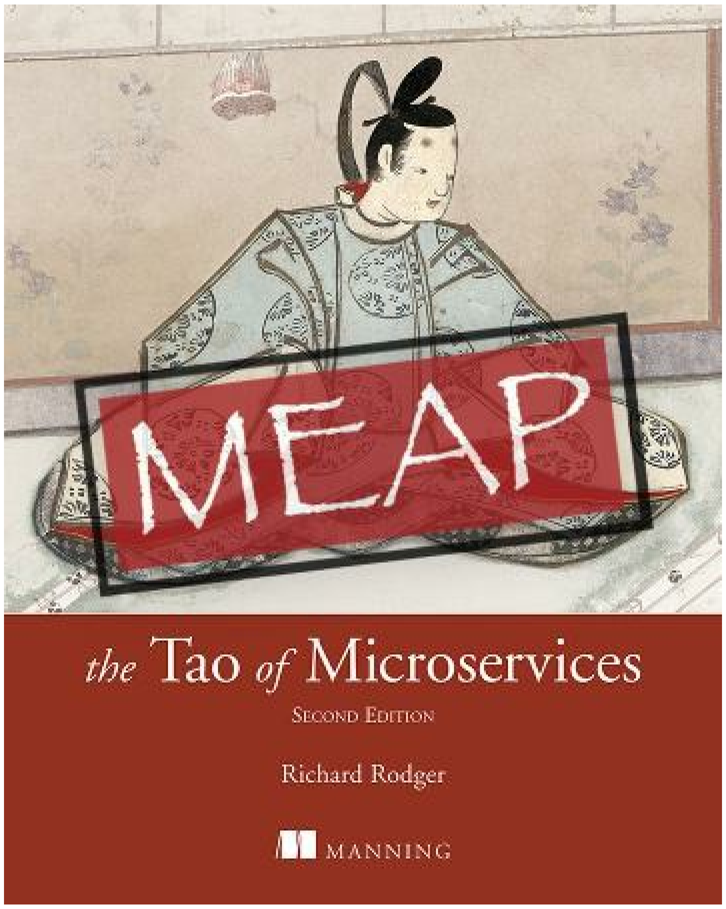
the Tao of Microservices

SECOND EDITION

Richard Rodger

MANNING

The Tao of Microservices, Second Edition MEAP V02



MEAP VERSION 2

MANNING PUBLICATIONS

Welcome

Thank you so much for purchasing *The Tao of Microservices 2nd Edition*! This book will help you to understand microservices, and more importantly, give you the ability to use them in production systems. I am an optimist, and I believe that things are getting better all the time. With each generation of software developers, our craft gets a little better, one step at a time. Microservices are just another stepping stone on this path. In this 2nd edition you'll benefit from my experiences taking the ideas in this book to the next level, and learn about a new technique I like to call "model-as-code".

This book is not about code (there isn't much). Nor is it about how to set up and configure any given cloud platform. It does not discuss frameworks or tools (much). For the code examples I use JavaScript, because that is what most people will be able to read. The ideas in this book work in any language. For the examples, I also use an Open Source microservices framework, of which I am the maintainer. You can use any framework, to none at all, and still apply the ideas in this book usefully. This is not a "how to" book, it is a "what to" book. This book will teach you what to do to make microservices work, and teach you how to think about them. The implementation details are always specific to your context.

Software development is both empowering and infuriating. In the words of the great Brian Kernighan, co-author of *The C Programming Language*;

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

And who can resist the temptation to be as clever as you can be? Our traditional programming tools, languages and platforms make it very easy to be clever. They give us all sorts of wonderful abstractions and language features, and our code quickly becomes gorgeously entangled. Not only impossible to debug, but also impossible to enhance. Modern software development in large organizations is so vulnerable to these complications, and we are so unable to calmly deliver features on time, that we have had to invent intricate ceremonial games and vast intellectual castles to use as protection from our failures.

It doesn’t have to be this way. Software development can become an engineering science. In the microservice architecture I see a small step towards the engineering and scientific mindset. We must accept the limitations of the human mind, understand our boundaries, and then apply that understanding to the way that we build software. The war is not with the machines, it is with ourselves.

Microservices are a software component model that works. They are a way of making big software systems out of little pieces. Little pieces of software are easy to understand, easy to reason about, and easy to inspect for correctness. The microservice architecture, and the principles that you use to compose these little pieces together, are the subject of this book. In some ways, you might consider this book a field report on what works and what does not. I have had the very special privilege to apply these principles for my clients, and in my own startup, over the last ten years, and to enjoy the wonderful results that they bring. I hope that you will have as much fun, and profit, writing microservices as I have had.

This book is organized into two parts. In the first part, we explore the basic principles, and the basic practicalities, of using microservices. How do you get started? What conceptual tools should you use for deciding which services to build? How should you model the system? How should you handle persistent data? Throughout this book, case studies, at various depths, are used to make the ideas concrete. Some of the case studies also include source code, so that you can really understand how things fit together.

In the second part of the book, we talk about how to get micro services into production. And not just in the technical sense. We tackle some of the politics you will face, and how to navigate legacy systems in large organizations. And once you are live, how you do stay up? We talk about how to monitor and manage your micro services.

The chapter topics are as follows:

Part I: Building Microservices

- Chapter 1 is a case study of a small microservices system, and introduces the book’s perspective on messages, services, components, and models.
- Chapter 2 is an overview of the microservice component model, and why it works.
- Chapter 3 introduces messages as first class citizens of the microservice architecture.
- Chapter 4 explores the fundamental properties of microservices.
- Chapter 5 introduces “model-as-code” - a new way to use an old idea to accelerate microservices projects.
- Chapter 6 takes a fresh look at data persistence, and how microservices do things differently.
- Chapter 7 is a case study that brings all the ideas in the preceding chapters together in a practical example.

Part II: Running Microservices

- Chapter 8 addresses the question of deploying and managing large numbers of microservices, the importance of immutability and security, and looks at enabling technologies like containers.
- Chapter 9 examines the techniques and theory of measuring microservices in production, an activity fundamental to their success.
- Chapter 10 guides you through the process of migrating a monolith into microservices.
- Chapter 11 focuses on people and organizations, and how the microservice architecture makes them work better.
- Chapter 12 is a comprehensive and detailed case study, warts and all, of a production microservice system. The nodezoo.com site is a search engine for Node.js modules, built using the microservice architecture, fully live and in production. An iteration-by-iteration approach is again used to make the experience of using microservices concrete, and full source code is available online.

What is the difference between Alchemy and Chemistry? The scientific method? No. It was open collaboration between scientists using a common language. In 1787 Antoine Lavoisier published *Méthode de nomenclature* *chimique*, an organized, systematic method for naming chemical elements. For the first time, chemists could communicate effectively with each other. General acceptance of the existence of the element Oxygen, and it's power to explain fire, followed quickly as the first major breakthrough enabled by this common language. As software developers, we live in the dark ages, and we have yet to experience our own enlightenment. This book is a report from the field, an attempt to build a common language, and an invitation to shared conversation.

If you have any questions, comments, or suggestions, please share them in Manning's liveBook's Discussion Forum for my book.

Thank You!

—Richard Rodger

In this book

1 Getting started: Building a microblogging startup

The best way to learn the theory of something is to see that theory put into practice. This book focuses on the theory of the microservices approach to system construction. So let’s get started with a practical example.

We shall design and model a microblogging system [1]. You will see declarative code and diagrams that define the system in a rigorous way. In this book we avoid the particulars of any given microservice platform or tooling. We will use JavaScript for code examples, because that is the language most developers have some knowledge of. You will be able to apply the ideas in this book to your own system whatever language and platform you use.

We will design the initial implementation of a microblogging startup, cut down to the bare functional essentials. We will design down to just above the level of implementation code (the full source is available online). You will start to learn the way of thinking about microservices explored in this book.

1.1 Messages First

How do you design software? How do you design anything? A design is a map that is to be made into reality. A good design, as a map, highlights the important features of that yet-to-be reality, and guides the builder.

To design software is to decide what it does. We can start from the informal requirements, and pull out actions that the software should perform. You may prefer to start with the data that the software operates on, and you may find that this improves your choice of the system’s actions. That is a perfectly fine approach. We will start from the actions here, and turn (or return) to the data later.

The actions of the software system can be represented as messages passed between the components of the system. Let us deliberately leave the components to later. Traditional software design (and especially object-oriented design) often starts with an outline of the components. This book rejects that design tactic as premature. It is better to start with the messages, and work out what components you need later. The messages can be directly derived from the informal requirements because they encode behavior, rather than concepts.

1.1.1 Convert Informal Requirements into Messages

Informal requirements are not code. That is their defining quality. They can be very formal very long specification documents. They can be written on the back of a napkin. They can be a neat set of agile "tasks" with accurately estimated "effort" points. They can be hand-written notes from a meeting with a senior Vice President.

In all cases, we want to boil the requirements down into things the system does. In the case of our microblogging startup, the key feature is that you can follow another user and read the blog entries that they have posted. Right there we can see when need a "post" action and a "follow" action, at least.

We need to represent these actions. Perhaps we should give them names? That would be a mistake. We don’t yet know if these are the best choice of actions in our system. We don’t know how they might need to be refactored, or what special cases might appear. We just don’t know enough about the problem at this stage to go around giving out arbitrary names to core elements of the design.

Instead, let’s just focus on representing these actions as messages containing some data. To post a blog entry, you’ll probably need to know who the posting user is, and you’ll need the content of their post. To follow a user, you’ll need to specify who the user is, and who they want to follow.

Let’s represent these messages using JSON.

Listing 1.1. Using JSON to encode the messages

```
// Posting an entry
```

```
{
  "post": "entry",
  "user": "rabbit",
  "text": "I'm late, I'm late, I'm late!"
}

// Following a user
{
  "follow": "user"
  "user": "alice",
  "target": "rabbit"
}
```

The data properties `post:entry` and `follow:user` indicate what the action of the message should be. They look like names for the messages (which we just abandoned!), but they are not. They do function like names in this very simple scenario. As your system grows, and you discover many edge-cases, and as you refactor and split your microservices, you’ll find that the ability to add additional and more fine-grained properties to your messages to be a life-saver. Messages and services should not be tightly coupled, and messages should not be tied to named pieces of infrastructure (such as queues) or code (such as functions).

This book will teach you how to keep messages and services decoupled. Refactoring and scaling your systems, and avoiding the dreaded "distributed monolith", will be much easier as a result.

[1] Imagine you’re launching Twitter in 2006.

1.2 Using Models

We can use the messages we have extracted from the informal requirements to start discovering the services that we need to build. We can group the messages in ways that make sense to us, and use each group to define a service that will contain the implementation code that can handle that group of messages.

This grouping of messages is not fixed, and can change over time as requirements change, or as we develop new insights that suggest refactorings that will give us a better structure for the system. This book does not prescribe a specific procedure for grouping messages into services. That is where your good judgment and experience come into play. This book does strongly suggest a set of design principles that make mixing and matching messages and services much easier.

The implementation of your services will require you to write code. Writing code means you start needing to make decisions about all sorts of foundational things. Where does your configuration live? Which data stores will you use? Where will you deploy your application? How will people in marketing update the text of user messages? How can you keep all the little things consistent, like the entries in drop-down lists?

There is also the data model. What data entities will you use, and what fields will they have? How will you do validation? How can you keep that validation consistent between local apps, web frontends, and many backend microservices. What about the messages themselves? Will you validate them in some way? Will you define message schemas?

1.2.1 Building a shared language

There is a school of software design known as "Domain Driven Design". It has some nice ideas we can adopt to help answer all those questions. This book does not prescribe the use of any specific methodology. You need to carefully choose working practices that will be effective for you and your team in the house that you live in.

One of the nice ideas from Domain Driven Design is the concept of a "ubiquitous language". The advice is that you should deliberately and carefully define the meanings of the specific terms that everybody on the project, technical and non-technical, will use. This shared language prevents a great many problems and keeps the requirements accurate and understandable. The insight is that you can only really do this inside a "bounded context". Once the world gets too big you should instead start translating between such languages.

A project using the microservice architecture is going to operate within a bounded context with one ubiquitous language. This works up to several hundred services and data entities [2]. Beyond that, you will need to split things up into separate contexts with their own internal languages.

In this book we take things a step further. One of the most useful developments in recent years has been the adoption of infrastructure-as-code tools. These tools let you define the machines, servers, databases, and all sorts of other supporting elements, of your system, using code. The infrastructure of the system is defined in a formal way, and you can build verifications and automations on top of this. The important thing is that code is used. Code is powerful, and we already have many tools to work with code.

There is a recurring fashion in software development for model-driven architectures. The big idea is that you need to design the system at a level above code, and then produce the code from that design. If you are a working programmer you are now free to laugh out loud. Us working programmers know that a sufficiently detailed description of the system is exactly the code that we end up writing, not some near-useless model written by architects who have forgotten how to code!

The idea of being model-driven, and somehow avoiding the complexity of real code, is very compelling. Every decade tries something new. First we had COBOL, BASIC, and SQL. Then we had 4GL languages. Then we had UML. Now we have "no-code" tools. Some have been more successful than others.

When you build microservices, the need to use a shared language, and to have some sort of formal model of the system, in code, becomes more acute. One of the many valid criticisms of the microservice approach is that it generates far too much busy-work. It’s hard to keep hundreds of services consistent. A shared language, expressed in a formal model, greatly reduces this cost.

A model-driven architecture, where the model is still code, turns out to be quite useful, in the same way that infrastructure-as-code has been.

Using a formal model

This book recommends that you use a formal model to define a shared language for your microservices project. This shared language defines all the "things" in your system, and ranges from infrastructure configuration, through to specifying the grouping of messages assigned to services, to the definition of your data entities and fields, and includes all the little things like the contents of drop-down lists.

You can provide this model to your microservices as a single hierarchical data structure. In practice, a single, relatively large JSON document will do. This is how you keep everything consistent.

You could edit the JSON document by hand, but the real power comes from treating it as a compilation target. You should pick your favorite configuration language or tool and use it generate the shared model as a JSON document. Your configuration tool should use a well-defined language, let you split your model code into multiple files, allow you validate the configuration (if the language is type-safe, even better), and provide you with all the facilities of a real programming language. Some tools even provide hot-reloading.

This model of the system, in code, gives you most of the benefits of the model-driven architecture, without much of the fuss. Let’s try this out with our microblogging system. Let’s maintain a list of messages, and a list of services, and define which service handles which message. This model will make it much easier to generate message queue definitions, service discovery configuration, and even automate the production of lovely architecture diagrams (which are always correct!).

We’ll add a few extra messages—maybe be the user can post videos too? And you’ll want to unfollow users as well as follow them. To keep things simple for now, we’ll continue to code the model in JSON, but relax the syntax rules to reduce wear and tear on our eyeballs.

Listing 1.2. Defining a Model of the System

```
// Define the messages in the system
messages: {

  // The messages are defined using `name:value` pairs where the
  // `name` is the top level, and the `value` is the next level.
  // This lets us avoid repetition, and allows for meta data.
  post: {
    entry: { /* Message meta data goes here. */ },
    video: {},
  },

  follow: {
    user: {}
  },

  unfollow: {
    user: {}
  },
}

// Define the services, and which message groups they will accept
services: {

  // The top level is the name of the service, in this case `post
  post: {

    // The `in` property defines the messages for this service.
    in: {
      post: {}
    }
  },

  // The `follow` service.
  follow: {
    in: {
      follow: {},
      unfollow: {}
    }
  },
}
```

To specify messages without having to repeat all the properties, this structure uses only the message properties that are currently considered significant. Thus, for the `post:entry` message, the `user` and `text` properties are considered to be just parameters, so can be ignored for the purposes of routing messages to the right service.

All messages that contain a property `post` with value `entry` should end up at the `post` microservice. All messages that have a property `follow` or `unfollow`, with any value, should end up at the `follow` microservice.

There's nothing special about this structure. You can make up your own structure! You can decide to use strict formal methods, or keep things simple and lenient. The important thing is that you can validate and share it. The model can be extended to cover any part of the system that needs to stay consistent between microservices.

With our messages and model in hand, showing how we have grouped messages by the services that will handle them, we have now drawn enough of a map to guide us in building the system.

1.2.2 Services from Messages and Models

We'll use case studies throughout this book to illustrate practical applications of the microservice approach. We'll use this first one to introduce some of the core principles of the architecture before we analyze them more deeply. The case studies will keep you connected to the practical side of the discussion and allow you to make a critical assessment of the ideas presented —would you build things the same way?

In this case study, you're building the *minimum viable product* (MVP)[3] for a new startup. The startup has come up with a crazy new idea called *microblogging*. Each blog entry can be only a few paragraphs long, with a maximum of 1,729 characters. This somewhat arbitrary limit was chosen by the founders as the most uninteresting number they could think of. The startup is called http://ramanujan.io.[4] It may seem strange to use a startup as a case study when our primary concern is enterprise software development. But isn't the goal to be as nimble as a startup?

We'll follow the startup through a series of iterations as it gets the MVP up and running. It's sometimes said that microservices create too much overhead at the start of a project. That misses the primary benefit of microservices— the ability to add functionality quickly!

Iteration 0: Posting entries

This is the first iteration. We’ll use iterations to follow the story of the startup.

A microblogging system lets users post *entries*: short pieces of text. There’s a page where users can view their own entries. This set of activities seems like a good place to start.

What activities happen in this part of the system?

- Posting an entry
- Listing your previous entries

There are all sorts of other things, like having a user account and logging in, that we’ll ignore for the sake of keeping the case study focused. These activities are amenable to the same style of analysis.

The activities can be represented by messages. No need to over-think the structure of these messages, because you can always change the mapping from message to microservice later. To post an entry, let’s have a message that tells you which user is posting the entry, and the text of the entry. You’ll also need to classify the message in some way so that you know what sort of message it is. The property-value pair `post:entry` does this job—a little namespacing is always a good idea. Let’s use JSON as the data format for messages in our implementation. For our examples in this book, we’ll continue to use JavaScript-style syntax as that is easier to read.

Listing 1.3. Posting an entry

```
{
  post: 'entry',
  user: 'alice',
  text: 'Curiouser and curiouser!'
}
```

Any interested microservices can recognize this message by looking for the pattern `post:entry` in the message’s top-level properties. For now, you’ll assume that messages can make their way to the right microservice without worrying too much about how that happens. (Chapter 3 has much more to say about message routing.)

You also need a message for listing entries. Standing back for a moment, you can assume that the system may include other kinds of data down the road. There are certainly common operations that you'll perform on data entities, such as *loading*, *saving*, and *listing*. Let's add a `store` property to the message to create a namespace for messages concerned with persistent data. In this case, you want to list things from the data store, so the property-value pair `store:list` seems natural. You'll use `kind:entry` to identify the data entity as an *entry*, assuming you'll have other kinds of data entities later.

Listing 1.4. Listing entries for the user *alice*

```
{
  store: 'list',
  kind: 'entry',
  user: 'alice'
}
```

Time to put on your architecture hat. There's a family of data-operation messages here, with a corresponding set of patterns:

- `store:list,kind:entry`—Lists entries, perhaps with a query constraint on the result list
- `store:load,kind:entry`—Loads a single entry, perhaps using an `id` property in the message
- `store:save,kind:entry`—Saves an entry, creating a new database row if necessary
- `store:remove,kind:entry`—Removes an entry from the database, using the `id` property to select it

This is an initial outline of the possible data-operation message patterns. This set of properties feels workable, but is it correct? It doesn't matter. You can always change the patterns later. Also, you don't need to implement messages you aren't using yet.

Now that you have some initial messages, you can think about their interactions. Let's assume you have a web server handling inbound HTTP requests on the external side and generating microservice messages on the internal side:

- When the user posts a new entry, the web server sends a `post:entry` message, which triggers a `store:save,kind:entry` message.
- When the user lists their previous entries, the web server sends a `store:list,kind:entry` message to get this list.

Here's another thing to think about: are these messages *synchronous* or *asynchronous*? In more-concrete terms, does the sender of the message expect to get a response (synchronous), or does the sender not care about a response (asynchronous)?

- `post:entry` is *synchronous*, because it's nice for the user to get confirmation that their entry has been posted.
- `store:save,kind:entry` is also *synchronous*, because it has to provide confirmation of the *save* operation and probably returns the generated unique identifier of the new data record.
- `store:list,kind:entry` is necessarily *synchronous*, because its purpose is to return a result list.

Are there any asynchronous messages in this simple first iteration? As a rule of thumb, microservice systems often benefit from announcement messages. That is, you should let the world know that something happened, and the world can decide if it cares. This suggests another kind of message:

- `info:entry` is *asynchronous* and announces that a new entry has been posted. No reply is expected. There may be a microservice out there that cares, or maybe nobody cares.

These architectural musings allow you to tabulate the message flows for your two activities, as shown in table 1.1.

Table 1.1. Business activities and their associated message flows

Activity	Message flow
Post entry	1. `post:entry` 2. `store:save,kind:entry` 3. `info:entry`

List entry	1. `store:list,kind:entry`

You haven’t even thought about microservices yet. By thinking about messages *first*, you’ve avoided falling into the trap of working on the implementation before understanding what you need to build.

At this point, you have enough to go on, and you can group messages into sensible divisions, suggesting the appropriate microservices to build in this iteration. Here are the microservices:

- *front*—The web server that handles HTTP requests. It sits behind a traditional load balancer.
- *entry-store*—Handles persistence of entry data.
- *post*—Handles the message flow for posting an entry.

Each microservice sends and receives specific messages, which you can tabulate as shown in table 1.2. The diagram in figure 1.1 shows how this all fits together.

Table 1.2. Messages that each microservice sends and receives

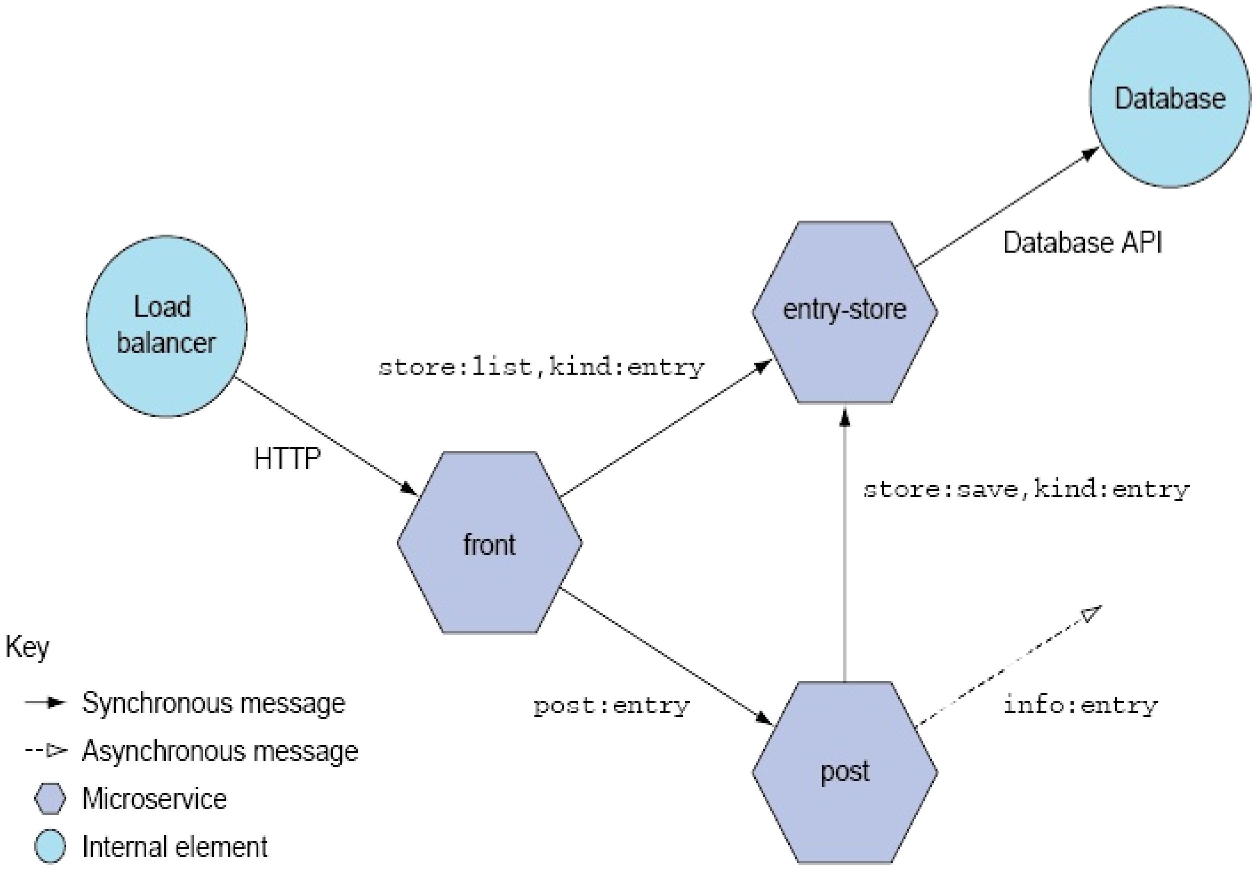
Microservice	Sends	Receives

front	`post:entry`
	`store:list,kind:entry`	

entry-store		`store:list,kind:entry`
		`store:save,kind:entry`

post	`store:save,kind:entry`
	`info:entry`
		`post:entry`

Figure 1.1. Iteration 0: Messages and services that support posting and listing entries



Here are the architectural decisions you've made:

- There's a traditional load balancer in front of everything.
- The web server is also a microservice (*front*) and participates in the message flows. It doesn't accept messages from external clients, only proxied HTTP requests from the load balancer.
- The *front* service should be considered the boundary of the system. It translates HTTP requests into internal messages.
- The *entry-store* microservice exposes a data store, but only via messages. No other microservice can access the underlying database [5].
- The *post* service orchestrates the message flow that implements posting an entry. First it performs the synchronous `store:save,kind:entry`; once it has a confirmed save, it emits an asynchronous `info:entry`.

This little microblogging system allows users to post entries and see a list of their previous entries. For now, assume deployment is fully automated, and driven by your model of the system—chapter 8 covers deployment of microservices. It's Friday, you've pushed code, and you can go home.

Iteration 1: A search index

In the last iteration, you used a method of system design that works very well for microservice architectures. First, you informally described the *activities* in the system. Then, you represented those activities as *messages*. Finally, you derived *services* from the messages. Messages, it turns out, are more important than services.

The task in this iteration is to introduce a search index so that users can search through entries to find wonderful gems of microblogging wit and insight amid a sea of sentiment. The system will use a search engine running inside the network to provide the search functionality. This suggests a microservice to expose the search engine, similar to the way the entry database is exposed by the *entry-store* microservice. We don't really want to expose our database and search engine directly to microservices. If we **mediate** these external elements of the system via messages, and our reward is flexibility—messages can go anywhere and be implemented by anything [6].

But we're getting ahead of ourselves. First, what are the messages? Users can add entries to the search engine, and they can query the search engine. That's enough for now. This gives you the following message patterns:

- `search:insert`—Inserts an entry
- `search:query`—Performs a query, returning a list of results

The *front* microservice can issue `search:query` messages to get search results. That seems OK. The *post* microservice can orchestrate a `search:insert` message into its entry-posting workflow. That doesn't seem OK. Didn't you introduce an `info:entry` asynchronous message for exactly the purpose of acting on new entries? The search engine microservice—let's call it *index*—should listen for `info:entry` and then insert the new entry into its search index. That keeps the *post* and *index* microservices decoupled (almost).

Something still isn't right. The problem is that the *index* microservice is concerned only with activities related to search—why should it know anything about posting microblogging entries? Why should *index* know that it has to listen for `info:entry` messages? How can you avoid this semantic coupling?

The answer is *translation*. The business logic of the *index* microservice shouldn't know about microblogging entries, but it's fine if the runtime configuration of the *index* microservice does.

The runtime configuration of *index* can listen for `info:entry` messages and translate them into `search:insert` messages locally. Loose coupling is preserved. The ability to perform this type of integration, without creating the tight coupling that accumulates technical debt, is the payoff you're seeking from microservices. The business logic implementing a microservice can have multiple runtime configurations, meaning it can participate in the system in many ways, without requiring changes to the business-logic code.

Let's represent this message translation in our model, to make it explicit, rather than hiding it inside an individual microservice configuration. We'll continue to assume that our deployment setup will magically wire things up in the right way.

Listing 1.5. Extending the Model to Describe Message Translations

```
...
services: {
  ...
  index: {
    in: {
      search: {
        query: {},
        insert: {}
      },
      info: {
        entry: {
          // Use $ as a suffix to mark meta data
          translate$: {
            remove: ['info'],
            add: { search: { insert: {} } }
          }
        }
      }
    }
  }
}
```

This little slice of the model says that the *index* service will accept any messages that contain `search:query`, `search:insert`, and `info:entry`. Also, `info:entry` has some meta data, a `translate$` directive, which adds and removes properties from the message.

The message `info:entry,text:'I wonder which way I ought to go?',user:alice` becomes: `search:insert,text:'I wonder which way I ought to go?',user:alice`. We'll assume the *index* service knows how to deal with the `text` and `user` properties.

What gives meaning to the `translate$` directive? You do! It is something you choose to recognize in your deployment system. Or perhaps you have a shared library that performs translation dynamically. The value of the model lies in the shared representation, the map, of the system. The sophistication of the infrastructure that uses the model is secondary. If the model drives a set of hacked-together shell scripts, that is still better than a system that only has hacked-together shell scripts. Chapter 5 will help you get much more comfortable with this idea.

At this stage you might be feeling that our model syntax is getting a little verbose. Maybe it is time to switch to an even easier format, such as YAML? That's not a bad idea, and you can certainly do this in your systems. In this book we'll stick with an abbreviated form of JSON so that we can continue to leverage JavaScript as a *lingua franca*. Here's a shorter version of the model code:

Listing 1.6. Abbreviated JSON for Writing System Models

```
// First Abbreviation - `a: b: 1` means { a: { b: 1 } }
// Second Abbreviation - `a: b: 1` and `a: c: 2`
<lineArrow></lineArrow>together mean { a: { b: 1, c: 2 } }
services: index: {
  in: search: query: {}
  in: search: insert: {}
  in: info: entry: {
    translate$: remove: ['info']
    translate$: add: search: insert: {}
  }
}
```

Let’s update our table to see where we are.

Table 1.3 shows the new list of services and their message allocations.

Table 1.3. Messages that each microservice sends and receives

Microservice	Sends	Receives

front	`post:entry`
	`store:list,kind:entry`
	`search:list`	

entry-store		`store:list,kind:entry`
		`store:save,kind:entry`

post	`store:save,kind:entry`
	`info:entry`
		`post:entry`

index		`search:query`
		`search:insert`
		`info:entry`

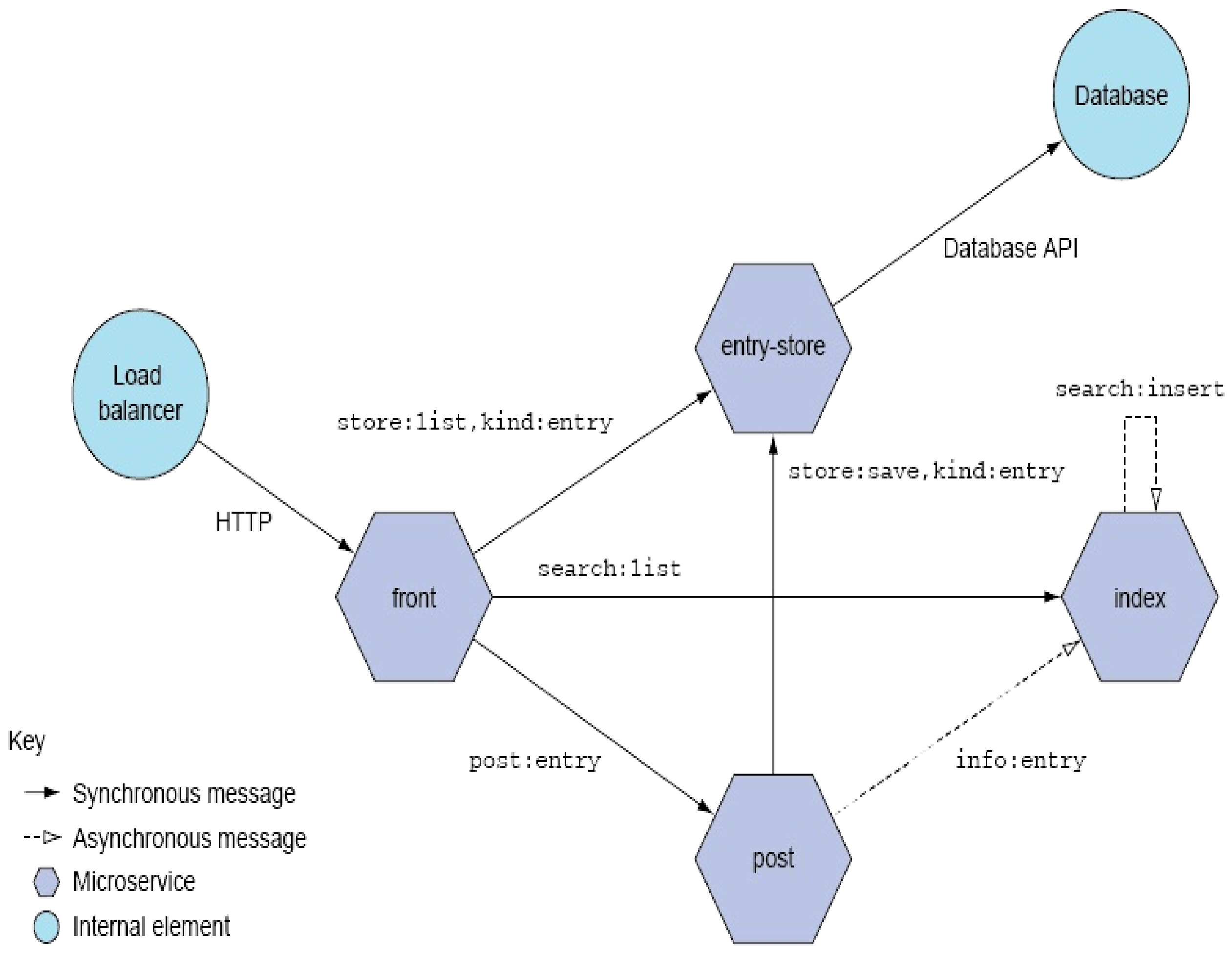
Now, you’ll also apply the principle of *additivity*. To your live system, running in production, you’ll deploy the new *index* microservice. It starts listening for `info:entry` messages and adding entries to the search index. This has no effect on the rest of the system. You review your monitoring system and find that clients still experience good performance and nothing has broken. Your only action has been to add a new microservice, leaving the others running. There’s been no downtime, and the risk of breakage was low.

You still need to support the search functionality, which requires making changes to an existing microservice. The *front* service needs to expose a search endpoint and display a search page. This is a riskier change. How do you roll back if you break something? In a traditional monolithic [7] architecture, teams often use a *blue-green configuration*, where two copies (blue and green) of the entire system are kept running at the same time. Only one is live; the other is used for deployments. Once validated, the systems are switched. If there's a glitch, you can roll back by switching the systems again. This is a lot of overhead to set up and maintain.

Consider the microservice case, where you can use the property of additivity. You deploy a new version of the *front* microservice—one that can handle searches—by starting one or more instances of the new version of *front* alongside running instances of the old version of *front*. You now have multiple versions in production, but you haven't broken anything, because the new version of *front* can handle entry posting and listing just as well as the old version. The load balancer splits traffic evenly between the new and old versions of *front*. You have the option to adjust this if you want to be extra careful and use the load balancer to send only a small amount of traffic to the new version of *front*. You get this capability without having to build it into the system; it's part of the deployment configuration.

Again, you monitor, and after a little while, if all is well, you shut down the running instances of the old version of *front*. Your system has gained a new feature by means of you adding and removing services, not via global modification and restart. Yes, you did "modify" the *front* service, but you could treat the new version as an entirely new microservice and stage its introduction into the live system. This is very different from updating the entire system and hoping there are no unforeseen effects in production. The new system is shown in figure 1.2.

Figure 1.2. Iteration 1: Adding messages and services that support searching entries



Consider table 1.4, which lists the series of small, safe deployment steps that got you here from the iteration 0 system. Another successful week! It's Friday, and you can go home.

Table 1.4. A sequence of small deployment steps

Step	Microservice/Version	Action
0	*index/1.0*	Add
1	*front/2.0*	Add
2	*front/1.0*	Remove

Iteration 2: Simple composition

Microservices are supposed to be components, and a good component model enables composition of components. Let’s see how this works in a microservice context. The *entry-store* microservice loads data from an underlying database. This operation has a relatively high latency—it takes time to talk to the database. One way to improve perceived performance is to decrease latency, and one way to do that is to use a cache. When a request comes in to load a given entry, you check the cache first, before performing a database query.

In a traditional system, you’d use an abstraction layer within the code base to hide the caching interactions. In particularly bad code bases, you may have to refactor first to even introduce an abstraction layer. As a practical matter, you have to make significant changes to the logic of the code and then deploy a new version of the entire system.

In our little microservice architecture, you can take a different road: you can introduce an *entry-cache* microservice that captures all the messages that match the pattern `store:*,kind:entry`. This pattern matches all the data-storage messages for entries, such as `store:list,kind:entry` and `store:load,kind:entry`. The new microservice provides caching functionality: if entry data is cached, return the cached data; if not, send a message to the *entry-store* service to retrieve it. The new *entry-cache* microservice captures messages intended for the existing *entry-store*.

There’s a practical question here: how does message capture work? There’s no single solution, because it depends on the underlying message transportation and routing.

One way to do message capture is to introduce an extra property into the `store:*` messages—say, `cache:true`. You tag the message with a property that you’ll use for routing. Then, you deploy a new version (2.0) of the *entry-store* service that can also listen for this pattern. By “new version,” I mean only that the runtime message-routing configuration has changed. Then, you deploy the *entry-cache* service, which listens for `store:*` as well, but sends `store:*,cache:true` when it needs original data:

- *entry-store*—Listens for `store:*` and `store:*,cache:true`
- *entry-cache*—Listens for `store:*`, and sends `store:*,cache:true`

The other services then load-balance `store:*` messages between these two services [8] and receive the same responses as before, with no knowledge that 50% of messages are now passing through a cache.

Finally, you deploy another new version (3.0) of *entry-store* that only listens for `store:*,cache:true`. Now, 100% of `store:*` messages pass through the cache:

- *entry-store*—Listens for `store:*,cache:true` only
- *entry-cache*—Listens for `store:*`, and sends `store:*,cache:true`

You added new functionality to the system by adding a new microservice. You did *not* change the functionality of any existing service.

Table 1.5 shows the deployment history.

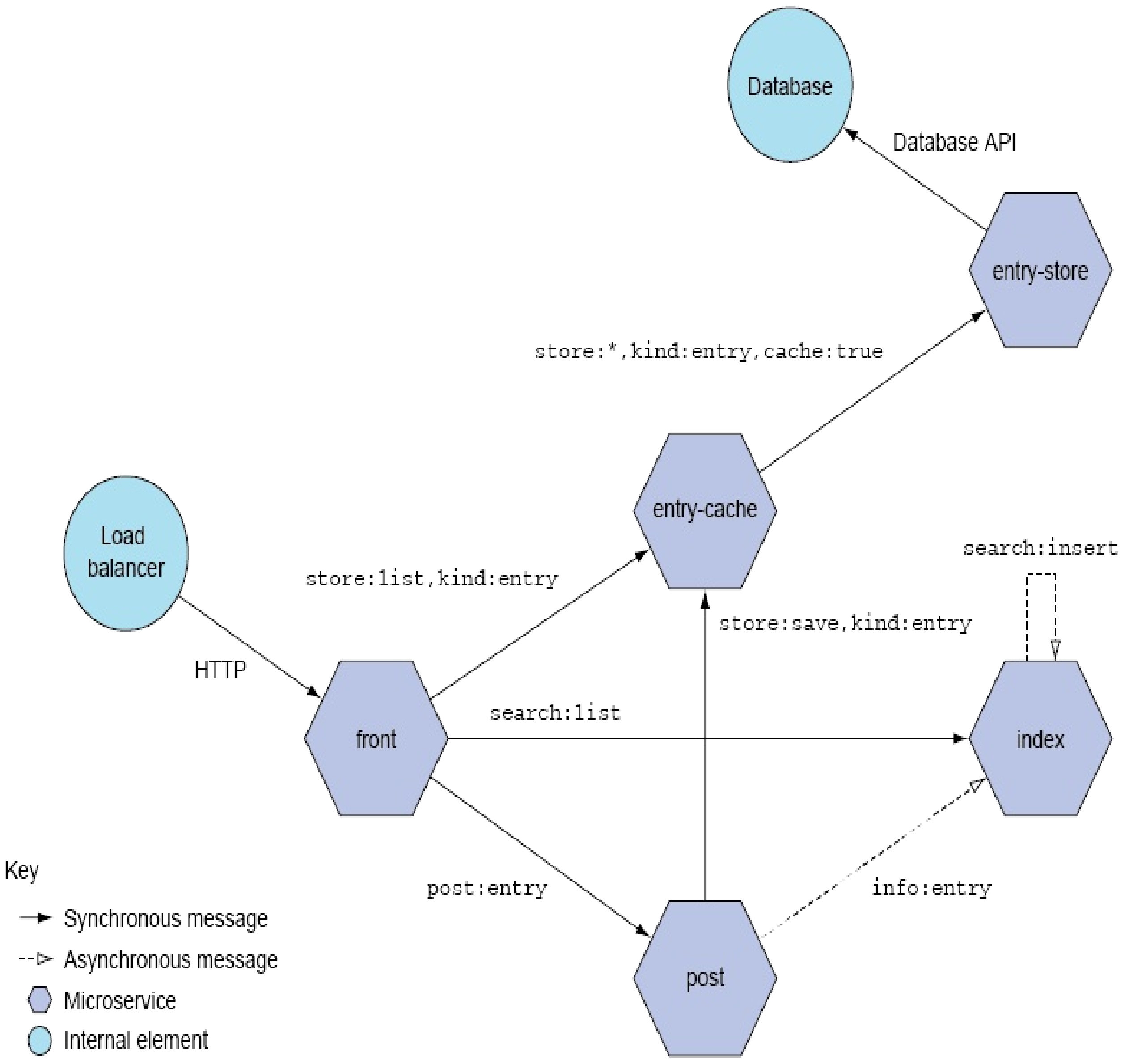
Table 1.5. Modifications to the behavior of *entry-store* over time

Step	Microservice/Version	Action	Message patterns
0	*entry-store/2.0*	Add	`store:*` and `store:*,cache:true`
1	*entry-store/1.0*	Remove	`store:*`
2	*entry-cache/1.0*	Add	`store:*`
3	*entry-store/3.0*	Add	`store:*,cache:true`

4	*entry-store/2.0*	Remove	`store:*` and `store:*,cache:true`

You can see that the ability to perform deployments as a series of add and remove actions gives you fine-grained control over your microservice system. In production, this ability is an important way to manage risk, because you can validate the system after each add or remove action to make sure you haven't broken anything. Figure 1.3 shows the updated system.

Figure 1.3. Iteration 2: An entry store cache implemented by message capture



The message-tagging approach assumes you have a transport system where each microservice can inspect every message to see whether the message is something it can handle. For a developer, this is a wonderfully useful fiction. But in practice, this isn’t something you want to do, because the volume of network traffic, and the work on each service, would be too high. Just because you can assume universal access to all messages as a microservice developer doesn’t mean you have to run your production system this way. In the message-routing layer, you can cheat, because you already know which messages which microservice cares about—you specified them in your model!

You’ve *composed* together the caching functionality of *entry-cache* and the data-storage functionality of *entry-store*. The rest of the world has no idea that the `store:*, kind:entry` messages are implemented by an interaction of two microservices. The important thing is that you were able to do this without exposing internal implementation details, and the microservices interact only via their public messages.

This is powerful. You don’t have to stop at caching. You can add data validation, message-size throttling, auditing, permissions, and all sorts of other functionality. And you can deliver this functionality by composing microservices together at the component level. The ability to do fine-grained deployment is often cited as the primary benefit of microservices, but it isn’t. The primary benefit is composition under a practical component model (that is, you can add functionality quickly without breaking things).

Another successful week.

Iteration 3: Timelines

The core feature of a microblogging framework is the ability to follow other users and read their entries. Let’s implement a Follow button on the search result list, so that if a user sees somebody interesting, they can follow that person. You’ll also need a home page for each user, where they can see a timeline of entries from all the other users they follow. What are the messages?

- `follow:user`—Follows somebody
- `follow:list,kind:followers|following`—Lists a user’s followers, or who they’re following
- `timeline:insert`—Inserts an entry into a user’s timeline
- `timeline:list`—Lists the entries in a user’s timeline

This set of messages suggests two services: *follow*, which keeps track of the social graph (who is following who), and *timeline*, which maintains a list of entries for each user, based on who they follow.

You aren’t going to extend the functionality of any existing services. To add new features, you’ll add new microservices. This avoids technical debt by moving complexity into the message-routing configuration and out of conditional code and intricate data structures.

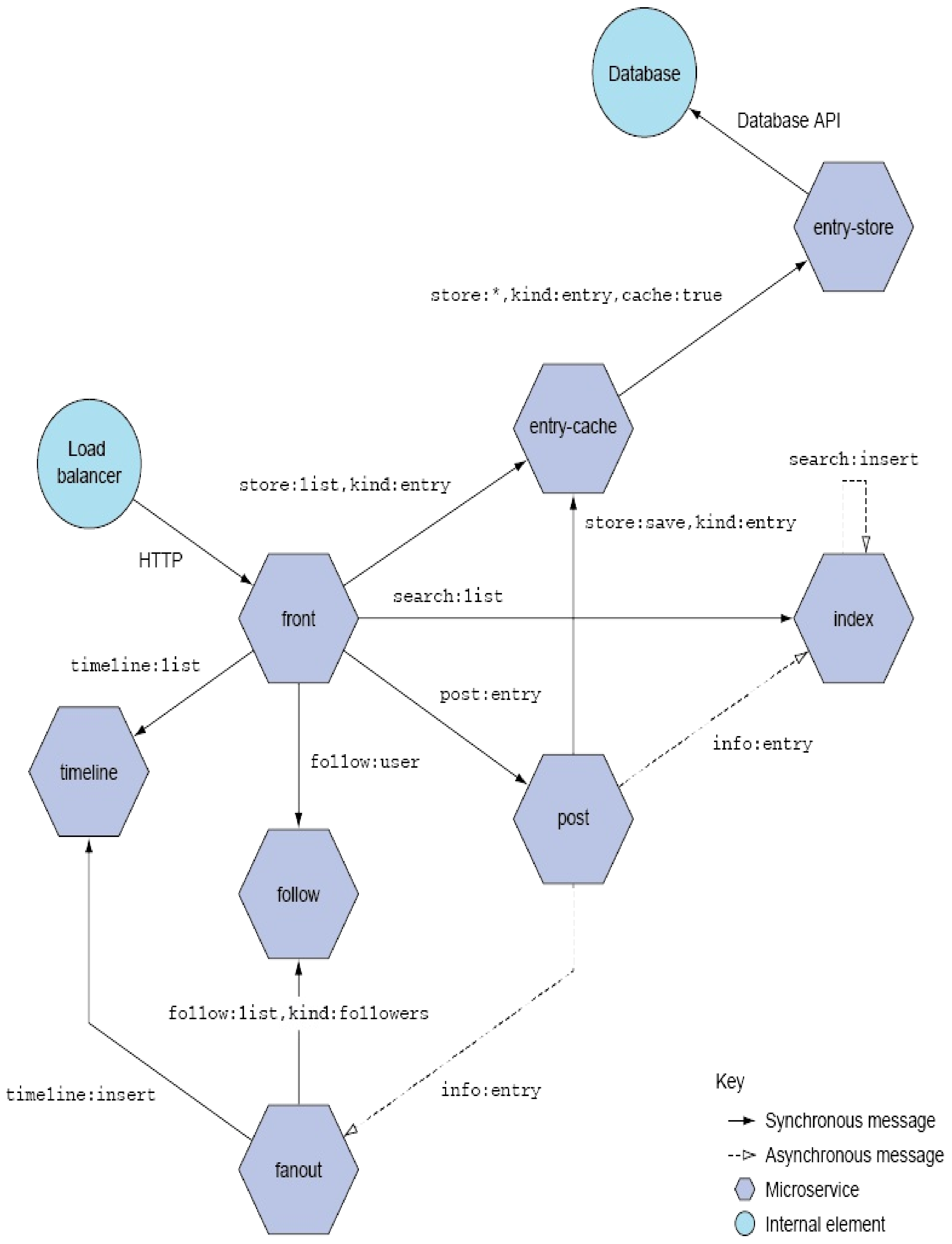
Using pattern matching to route messages can handle this complexity more effectively than programming language structures because it’s a *homogeneous* representation of the business activities you’re modeling. The representation consists *only* of pattern matching on messages and using these patterns to assign them to microservices. Nothing more than simple pattern matching is needed. You understand the system by organizing the message patterns into a hierarchy, which is much easier to comprehend than an object-relationship graph.

At this point, you face an implementation question: should timelines be constructed in advance or on demand? To construct a timeline for a user on demand, you’d have to get the list of users that the user follows—the user’s “following” list. Then, for each user followed, you’d need to get a list of their entries and merge the entries into a single timeline. This list would be hard to cache, because the timeline changes continuously as users post new entries. This doesn’t feel right.

On the other hand, if you listen for `info:entry` messages, you can construct each timeline in advance. When a user posts an entry, you can get the list of their followers; then, for each follower, you can insert the entry into the follower’s timeline. This may be more expensive, because you’ll need extra hardware to store duplicate data, but it feels much more workable and scalable.[9] Hardware is cheap.

Building the timelines in advance requires reacting to an `info:entry` message with an orchestration of the `follow:list,kind:followers` and `timeline:insert` messages. A good way to do orchestration is to put it into a microservice built for exactly that purpose. This keeps intelligence at the edges of the network, which is a good way to manage complexity. Instead of complex routing and workflow rules for everything, you understand the system in terms of the inbound and outbound message patterns for each service. In this case, let’s introduce a *fanout* service that handles timeline updates. This *fanout* service listens for `info:entry` messages and then updates the appropriate timelines. The updated system with the interactions of the new *fanout*, *follow*, and *timeline* services is shown in figure 1.4.

Figure 1.4. Iteration 3: Adding social timelines



Both the *follow* and *timeline* services store persistent data, the social graph, and the timelines, respectively. Where do they store this data? Is it in the same database that the *entry-store* microservice uses? In a traditional system, you end up putting most of your data in one big, central database, because that's the path of least resistance at the code level. With microservices, you're freed from this constraint. In the little microblogging system, there are four separate databases:

- The entry store, which is probably a relational database
- The search engine, which is definitely a specialist full-text search solution
- The social graph, which might be best handled by a graph database
- The user timelines, which can be handled by a key-value store

None of these database decisions are absolute, and you could certainly implement the underlying databases using different approaches. The microservices aren't affected by each other's choice of data store, and this makes changes easier. Later in the project, if you find that you need to migrate to a different database, then the impact of that change will be minimized.

At this point, you have a relatively complete microblogging service. Another good Friday for the team!

Iteration 4: Scaling

This is the last iteration before the big pitch for a series A venture capital round. You're seeing so much traction that you're definitely going to get funded. The trouble is, your system keeps falling over. Your microservices are scaling fine, because you can keep adding more instances, but the underlying databases can't handle the data volumes. In particular, the timeline data is becoming too large for one database, and you need to split the data into multiple databases to keep growing.

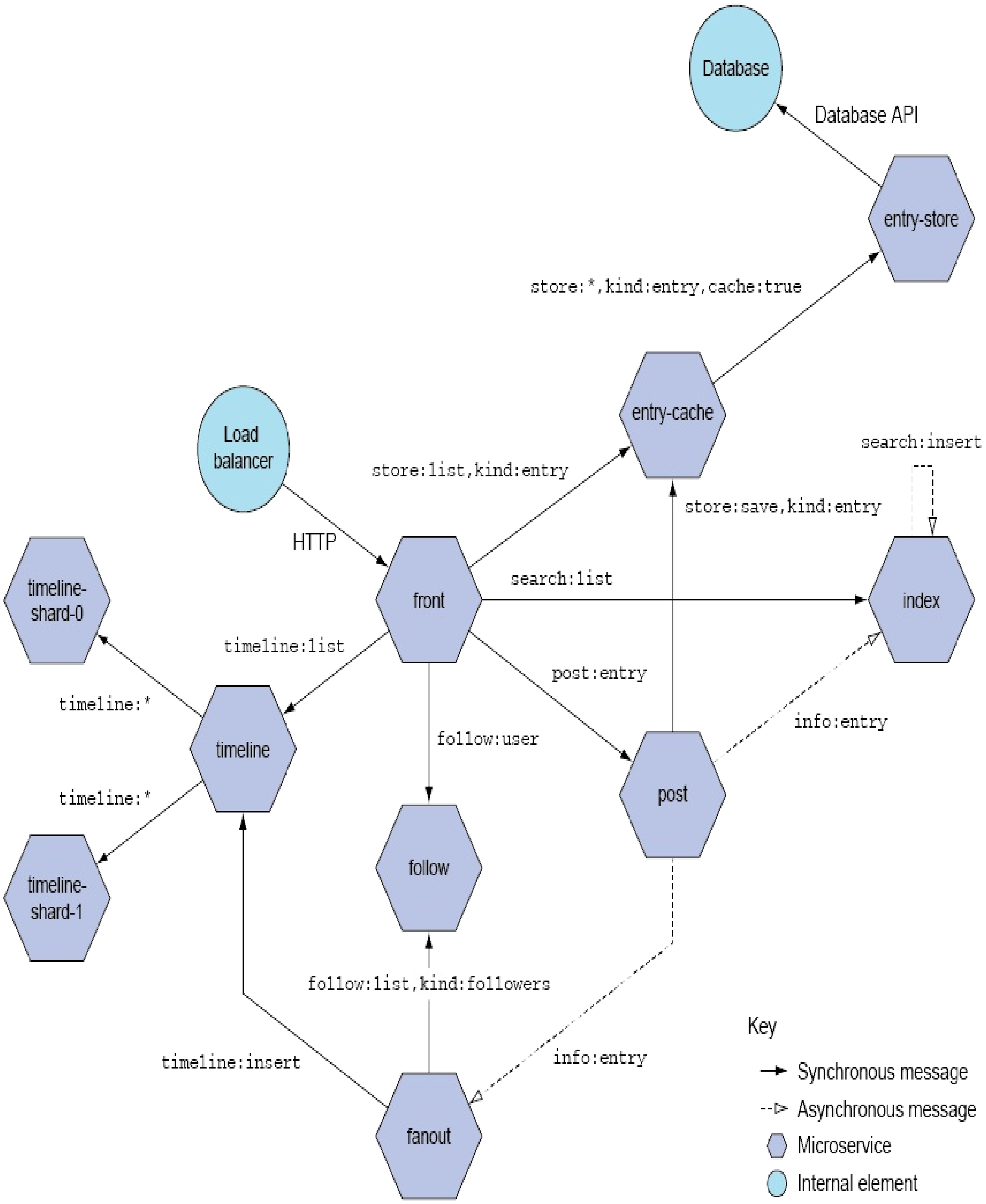
This problem can be solved with database *sharding*. Sharding works by assigning each item of data to a separate database, based on key values in the data. Here's a simplistic example: to shard an address book into 26 databases, you could shard on the first letter of a person's name. To shard data, you'd typically rely on the specific sharding capabilities of a database driver component, or the sharding feature of the underlying database. Neither of these approaches is appealing in a microservice context (although they will work), because you lose flexibility.

You can use microservices to do the sharding by adding a `shard` property to the `timeline:*` messages. Run new instances of the *timeline* service, one group for each shard, that react only to messages containing the `shard` property. At this point, you have the old *timeline* service running against the old database, and a set of new sharding *timeline* services. The implementation of both types of *timeline* is the same; you're just changing the deployment configuration and pointing some instances at new databases.

Now, you migrate over to the sharding configuration by using microservice composition. Introduce a new version of the *timeline* microservice, which responds to the old `timeline:*` messages that don't have a `shard` property. It then determines the shard based on the user, adds a `shard` property, and sends the messages onward to the sharded *timeline* microservices. This is the same structure as the relationship between *entry-cache* and *entry-store*.

There are complications. You'll be in a transitional state for a while, and during this period you'll have to batch-transfer the old data from the original database into the new database shards. Your new *timeline* microservice will need logic to look for data in the old database if it can't find it in the new shard. You'll want to move carefully, leaving the old database active and still receiving data until you're sure the sharding is working properly. You should probably test the whole thing against a subset of users first. This isn't an easy transition, but microservices make it less risky. Far more of the work occurs as careful, small steps—simple configuration changes to message routing that can be easily rolled back. It's a tough iteration, but not something that brings everything else to a halt and requires months of validation and testing.[10] The new sharding system is shown in figure 1.5.

Figure 1.5. Iteration 4: Scaling by sharding at the message level



Now, let’s assume that everything goes according to plan, and you get funded and scale up to hundreds of millions of users. Although you’ve solved your technology problems, you still haven’t figured out how to monetize all those users. You take your leave as you roll out yet another advertising play—at least it’s quick to implement using microservices!

Wouldn’t it be great if enterprise software development worked like this? The vast majority of software developers don’t have the freedom that startup software development brings, not because of organizational issues, but because they’re drowning in technical debt. That’s a hard truth to accept, because it requires honest reflection on our effectiveness as builders of software.

The microservice architecture offers a solution to this problem based on sound engineering principles, not on the latest project management or technology platform fashions. Yes, there are trade-offs, and yes, it does require you to think in a new way. In particular, the criticism that complexity is being moved around, not reduced, must be addressed. It isn’t so. Message flows (being one kind of thing) are more understandable than the internals of a monolith (being many kinds of things). Microservices interact only by means of messages and are completely defined by these interactions. The internal programming structures of monoliths interact in all sorts of weird and wonderful ways, and offer only weak protection against interference with each other.

A microservice system can be fully understood at an architectural level by doing the following:

1. Listing the messages. These define a language that maps back to business activities.
2. Listing the services and the messages they send and receive.

On a practical level, yes, you do need to define and automate your deployment configuration. That is a best practice and not hard to do with modern tools. And you need to monitor your system—but at the level of messages and their expected flows. This is far more useful than monitoring the health of individual microservice instances.

The case study is concluded!

[2] Since we do not have much of a science of software engineering, and very

little real experimental data, almost all software engineering literature, this book included, must make general statements that are supported only by the empirical observations, careful reasoning, and, relevant readings of the author. No claims in this book are asserted to be universal laws. They are rules of thumb, guidelines that have seemed to hold over many years of a career, or wisdom imparted from those who came before. I beg your forgiveness for making direct statements. These are, in all cases, merely hypotheses. But I am happy to defend them and put them in to practice when there is real client money at stake.

[3] The MVP is a product-development strategy developed by Eric Ries, a founder of the IMVU chat service (founded 2004): build only the minimum set of features that lets you validate your assumptions about a market, and then iterate on those features and assumptions until you find a product that fits the market.

[4] You can find the full source code and a workshop at https://www.manning.com/books/the-tao-of-microservices and http://ramanujan.io.

[5] For the moment. In time this microservice may be refactored away. Perhaps you'll have one microservice for all data entities, to ensure a common permissions system. Perhaps you'll extract data persistence into a shared library make `store:*` messages entirely local to each microservice.

[6] Hiding things behind messages makes unit testing much easier—mocking does not require replicating complex object structures, or standing up local instances of services. Instead, you hard-code the message responses.

[7] The term *monolith* stands for enterprise systems with large volumes of code that run as one process.

[8] Let's assume this "just works" for now. The example code at www.manning.com/books/the-tao-of-microservices and http://ramanujan.io has all the details, of course.

[9] Timeline insertion is how the real Twitter works, apparently. A little bird

told me.

[10] Sharding using microservices is by no means the “best” way. That depends on your own judgment as an architect for your own system. But it’s possible, using only message routing, and it’s a good example of the flexibility that microservices give you.

1.3 Summary

- To design a microservice system, start by translating the informal requirements into well-defined messages.
- When you have well-defined messages, you can assign them to microservices, and change this assignment as your system evolves.
- It’s a good idea to build a formal model as a shared representation of your system. This model can be used to ensure consistency within the system.
- The microservice architecture provides a component model that provides a strong composition mechanism. This enables additivity, the ability to add functionality by adding new parts rather than modifying old ones.
- Instead of refactoring code, you can refactor message flows. This allows you to make big changes in small low risk steps.

2 Components

Software developers have been chasing the idea of software components since the first electron crossed a transistor gate, and mathematicians have been chasing components for much longer than that. None of us can quite agree on what software components are but we all know that we need them. They are the secret to building large systems.

Our mainstream programming languages are too fine-grained, and operate below the level that we need for large scale work. So we invent component models and design patterns for using these models, attempting to tame the complexity that breeds quickly in our code.

If microservices are to fulfill their promise they too must have a component model. We should think carefully about what that model should be, and what problems it shoud solve.

2.1 The technical debt crisis

The problem is that we can’t write software fast enough. We can’t write software that meets business needs and that is sufficiently accurate and reliable, within the time constraints set by the markets in which our companies operate. When requirements change in the middle of a project, it damages our architecture so badly that we spiral into a death march of hacks and kludges to force our data structures, concepts, and entity relationships into compliance. We try to refactor or rewrite, and that delays things even further. Our component models fall apart.

You might be tempted to blame the business itself for this state of affairs. Requirements are always underspecified, and they keep changing. Deadlines are imposed without regard to the complexity of the problem. Bad management wastes time and eats up development schedules. It’s easy to become cynical and blame these issues on the stupidity of others.

Such cynicism is self-defeating and naïve. The world of business is harsh, and markets are unforgiving. Our nontechnical colleagues have complex challenges of their own. It's time to grow up and accept that we have a problem: we can't write software fast enough.

But *why?*

Be careful. If there were a silver bullet, we'd already be using it. Take methodologies: we fight over them because there are no clear winners. Some methodologies certainly are better than others, in the same way that a sword is a better weapon than a dagger, but neither is of much use in the gun fight that is enterprise software development. Or take a best practice like unit testing, which feels like it's valuable. Just because something feels good doesn't mean it *is* good. Intuitions can be misleading. We have a natural tendency toward superstition, forcing development practices into Procrustean beds. [11] Few of our best practices have any measure of scientific validation.

The problem is that we don't know how to pay down technical debt. No matter how beautifully we crystallize our initial designs, they fracture against the changing nature of reality. We try to make our software perfect: perfectly correct and able to perfectly meet requirements. We have all sorts of perfection-seeking behavior, from coding standards, to type systems, to strict languages, to hard integration boundaries, to canonical data models. And yet we still end up with a legacy mess to support.

We're not stupid. We know that we have to cope with change. We use flexible data structures, with scope for growth (in the right places, if we're lucky). We have this thing called *refactoring*, which is the technical term for getting caught making bad guesses. At least we have a professional-sounding term for the time we waste rewriting code so that we can start moving forward again.

We have *components*, which are the machine guns of software architecture: force multipliers that let you build big things out of small things. You only have to solve each problem once. Object-oriented languages are trying to be component systems, and so are web services. [12] So was structured programming (a fancy name for dropping the `GOTO` statement). We have all these technical best practices, and we're still too slow. Components, in particular, should make us faster. Why don't they?

We haven’t been thinking about components in the right way for a long time in mainstream enterprise programming. [13] We can just about build library components to talk to databases, perform HTTP requests, and package up sorting algorithms. But these components are technical infrastructure. We’re not so good at writing reusable components that deliver *business logic*. Components like that would speed up development. We’ve focused on making our components so comprehensive in functionality that it’s difficult to compose them together; we have to write lots of glue code. By trying to cover too many cases, we make the simple ones too complex.

What is business logic?

In this book, *business logic* is the part of the functionality that’s directly specific to the business goal at hand. User-profile management is business logic. A caching layer isn’t.

Business logic is your representation of the processes of your business using the structures of your programming language. What is business logic in one system many not be in another. It can change within the same system over time. The term is suggestive, not prescriptive.

The thing that makes components work is *composition*: making big things out of small things. The component models that reduce your workload, such as UNIX pipes and functional programming, have this feature. You combine parts to create something new.

Composition is powerful. It works because it only does one thing: adds components together. You don’t modify components; instead, you write new ones to handle special cases. You can code faster because you never have to modify old code. That’s the promise of components.

Consider the state of the software nation. The problem is that we can’t code fast enough. We have this problem because we can’t cope with technical debt. We don’t work with an engineering mindset. We haven’t used scientific methods to validate our beliefs. As a solution, components should be working, but they aren’t. We need to go back to basics and create a component model that delivers practical composition. Microservices, [14] built the right way, can help do that.

[11] Procrustes was a son of the Greek god Poseidon. He took great pleasure in fitting his house guests into an iron bed. To make the guest fit, he would either amputate their feet or stretch them on a rack. A *Procrustean bed* is a behavior that doesn't serve its intended purpose; it only perpetuates a superstition. Insistence on high levels of unit-test coverage is the prime example in our industry. The coverage target and the effort to achieve it are seldom adjusted to match the value generated by the code in production.

[12] We even have fancy component systems that were designed from the ground up, like OSGi and CORBA. They haven't delivered composability. The Node.js module system is a relatively strong approach and makes good use of semantic versioning, but it's restricted to one platform and exposes all the nuts and bolts of the JavaScript language, not to mention all the supply chain security issues. UNIX pipes are about as good as it gets, if you're looking for something that's widely used.

[13] This isn't universally true. The functional language communities in particular treat composability as a first-class citizen. But consider that, whereas you can compose pretty much anything on the UNIX command line using pipes, functions aren't generically composable without extra work to make their inputs and outputs play nicely together.

[14] If you're looking for a definition of the term *microservice*, you're not going to find it in this book. We're discussing an approach to software architecture that has its own benefits and trade-offs. Substance counts more than sound bites.

2.2 How the monolith betrays the promise of components

When I talk about *monoliths* in this book, I mean large, object-oriented systems [15] developed within, and used by, large organizations. These systems are long-lived, under constant modification to meet ongoing business requirements, and essential to the health of the business. They're layered, with business logic in all the layers, from the frontend down to the database. They have wide and deep class hierarchies that no longer represent business reality accurately. Complex dependencies between classes and objects have arisen in response. Data structures not only suffer from legacy artifacts but must be twisted to meet new models of the world and are translated between representations with varying degrees of fidelity. New features must touch many parts of the system and inevitably cause regressions (new versions break previously working features).

The components of the system have lost encapsulation. The boundaries between components are shot full of holes. The internal implementation of components has become exposed. Too many components know about too many other components, and dependency management has become difficult. What went wrong? First, object-oriented languages offer too many ways to interfere with other objects. Second, every developer has access to the entire code base and can create dependencies faster than the poor architects can shoot them down. Third, even when it’s clear that encapsulation is suffering, there’s often no time for the refactoring necessary to preserve it. And the need for constant refactoring compounds the problem. This is a recipe for ballooning technical debt.

The components don’t deliver on reusability. As a consequence of losing encapsulation, they’re deeply connected to other components and difficult to extract and use again. Some level of reuse can be achieved at the infrastructure level (database layers, utility code, and the like). The real win would be to reuse business logic, but this is rarely achieved. Each new project writes business logic anew, with new bugs and new technical debt.

The components don’t have well-defined interfaces. Certainly, they may have strict interfaces with a great deal of type safety, but the interfaces are too intricate to be well defined. There’s a combinatorial explosion of possibilities when you consider the many ways in which you can interact with an object: construction dependencies, method calls, property access, and inheritance; and that’s not even counting the extra frills that any given language platform might give you. And to use the object properly, you need to build a mental model of its state, and the transitions of that state, giving the entire interface an additional temporal dimension.

Components don’t compose, betraying the very thing for which they were named. In general, it’s hard to take two objects and combine them to create enhanced functionality. There are special cases, such as inheritance and mixins, [16] but these are limited and now considered harmful. Modern object-oriented best practice is explicit: favor composition over inheritance. Keep an internal instance of what would have been your superclass, and call its methods directly. This is better, but you still need too much knowledge of the internal instance.

The problem with the object-oriented model is that universal composition is difficult to do well as an afterthought of language design. It’s much better to design it in from the start so that it always works the same way and has a small, consistent, predictable, stateless, easily understood implementation model. A compositional model has hit the bull’s-eye when it can be defined in a fully declarative manner, completely independent of the internal state of any given component.

Why is composition so important? First, it’s one of the most effective conceptual mechanisms we have for managing complexity. The problem isn’t in the machine; the problem is in our heads. Composition imposes a strict containment of complexity. The elements of the composition are hidden by the result of the composition and aren’t accessible. There’s no way to create interconnected spaghetti code, because composition can only build strict hierarchies. Second, there’s significantly less shared state. By definition, composed components communicate with each other via a stateless model, which reduces complexity. Peer communication between components is still subject to all the nastiness of traditional object communication. But a little discipline in reducing the communication mechanisms, perhaps by limiting them to message passing, can work wonders. Reducing the impact of state management means the temporal axis of complexity is much less of a concern. Third, composition is additive. [17] You create new functionality by combining existing functionality. You don’t modify existing components. This means technical debt inside a component doesn’t grow. Certainly, there’s technical debt in the details of the composition, and it grows over time. But it’s necessarily less than the technical debt of a traditional monolith, which has the debt of compositional kludges, the debt of feature creep within components, and the debt of increasing interconnectedness.

What is complexity?

You might say that the fraction 111/222 is somehow more complex that the fraction 1/2. [18] You can make that statement rigorous by using the *Kolmogorov complexity measure*. First, express the complex thing as a binary string by choosing a suitable encoding. (I realize I'm leaving entire industries, like recorded music, as an exercise for you to solve, but bear with me!) The length of the shortest program that can output that binary string is a numeric measure of the complexity of the thing. Before you start, choose some implementation of a universal Turing machine so the idea of a "program" is consistent and sensible.

Here are the two programs, in (unoptimized!) colloquial C. One prints the value of 111/222:

```
printf("%f", 111.0/222.0);
```

And the other print the value of 1/2:

```
printf("%f", 1.0/2.0);
```

Feel free to compile and compress them any way you like. It will always take more bits to express the values 111 and 222 than it will to express 1 and 2 (sorry, compiler optimizations don't count). Thus, 111/222 is more complex than 1/2. This also satisfies our intuition that fractions can be reduced to their simplest terms and that this is a less complex representation of the fraction.

A software system that combines elements in a small number of ways (additive composition) compared to a system that can combine elements in many ways (object orientation) grows in complexity far more slowly as the number of elements increases. If you use complexity as a proxy measure of technical debt, you can see that object-oriented monoliths are more vulnerable to technical debt. How much more vulnerable? Very much more! The total possible interactions between elements grows exponentially with the number of elements and the number of ways of connecting them.[19]

[15] The essential characteristic of a monolith isn't that a large body of code executes inside a single process. It's that the monolith makes full use of its

language platform to connect separate functional elements, thereby mortally wounding the composability of those elements as components. The object-oriented nature of these systems and the fact that they form the majority of enterprise software are mostly due to historical accident. There are other kinds of monoliths, but the object-oriented architecture has been such a pretender to the crown of software excellence, particularly in its broken promise to deliver reusable components, that it's the primary target of this book's ire.

[16] Inheritance was to be the primary mechanism of composition for object-oriented programming. It fails because the coupling between superclass and subclass is too tight, and it's tricky to subclass from more than one superclass. Multiple inheritance (alternatively, mixins) as a solution introduces more combinatorial complexity.

[17] A system is *additive* when it allows you to provide additional implementations of functionality without requiring changes in those that depend on you. For a technical description, see Harold Abelson, Gerald Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs* (MIT Press, 1996), section 2.4.

[18] The Mandelbrot Set fractal is a good example of low complexity. It can be generated from a simple recurrence formula on complex numbers: http://mathworld.wolfram.com/MandelbrotSet.html. For details on Kolmogorov complexity, start here: https://en.wikipedia.org/wiki/Kolmogorov_complexity.

[19] For the mathematically inclined, k^(n(n-1)/2), where k is the number of ways to connect elements and n is the number of elements. This model is a worst case, because the graph of object dependencies isn't quite as dense, but it's still pretty bad.

2.3 The microservice idea

Large-scale software systems are best built using a component architecture that makes composition both possible and easy. The term *microservice* captures two important aspects of this idea. The prefix *micro* indicates that the components are small, avoiding accumulation of technical debt. In any substantial system, there will be many small components, rather than fewer large ones. The root *service* indicates that the components shouldn't be constrained by the limitations of a single process or machine and should be free to form a large network. Components—that is, microservices—can communicate with each other freely. As a practical matter, communication via messages, rather than shared state, is essential to scaling the network.

That said, the heart of the microservice idea is bigger than these two facets. It's the more general idea that composable components are the units of software construction, and that composition works well only when the means of communication between components is sufficiently uniform to make composition practical. It isn't the choice of the mechanism of communication that matters. What matters is the simplicity of the mechanism.

The core axioms of the microservice architecture are as follows:

- No components are privileged (*no privilege*).
- All components communicate in the same simple, homogeneous way (*uniform communication*).
- Components can be composed from other components (*composition*).

From these axioms proceed the more concrete features of the microservice architecture. Microservices in practice are small, because the smaller a service is, the easier it is to compose. And if it's small, its implementation language matters less. In fact, a service may be disposable, because rewriting its functionality doesn't require much work. And pushing even further, does the quality of its code really matter? We'll explore these heresies in chapter 11.

Microservices communicate over the network using messages. In this way, they share a surface feature with service-oriented architectures. Don't be fooled by the similarity; it's immaterial to the microservice architecture what data format the messages use or the protocols by which they're transported [20]. Microservices are entirely defined by the messages they accept and the messages they emit. From the perspective of an individual microservice instance, and from the perspective of the developer writing that microservice, there are only messages arriving and messages to send. In deployment, a microservice instance may be participating in a request/response configuration, a publish/subscribe configuration, or any number of variants. The way in which messages are distributed isn't a defining characteristic of the microservice architecture. All distribution strategies are welcome without prejudice. [21]

The messages themselves need not be strictly controlled. It's up to the individual microservice to decide whether an arriving message is acceptable. Thus, there's no need for schemas or even validation. If you're tempted to establish contracts between microservices, think again. All that does is create a single large service with two separate, tightly coupled parts. This is no different from traditional monolithic architectures, except that now you have the network to deal with as well.[22] Flexibility in the structure of messages makes composition much easier to achieve and makes development faster, because you can solve the simple, general cases first and then specialize with more microservices later. This is the power of additivity: technical debt is contained, and changing business requirements can be met by adding new microservices, not modifying (and breaking) old ones.

A network of microservices is dynamic. It consists of large numbers of independent processes running in parallel. You're free to add and remove services at will. This makes scaling, fault tolerance, and continuous delivery practical and low risk. Naturally, you'll need some automation to control the large network of services. This is a good thing: it gives you control over your production system and immunizes you against human error. Your default operational action is to add or remove a single microservice instance and then to verify that the system is still healthy. This is a low-risk procedure, compared to big-bang monolith deployments.

2.3.1 The core technical principles

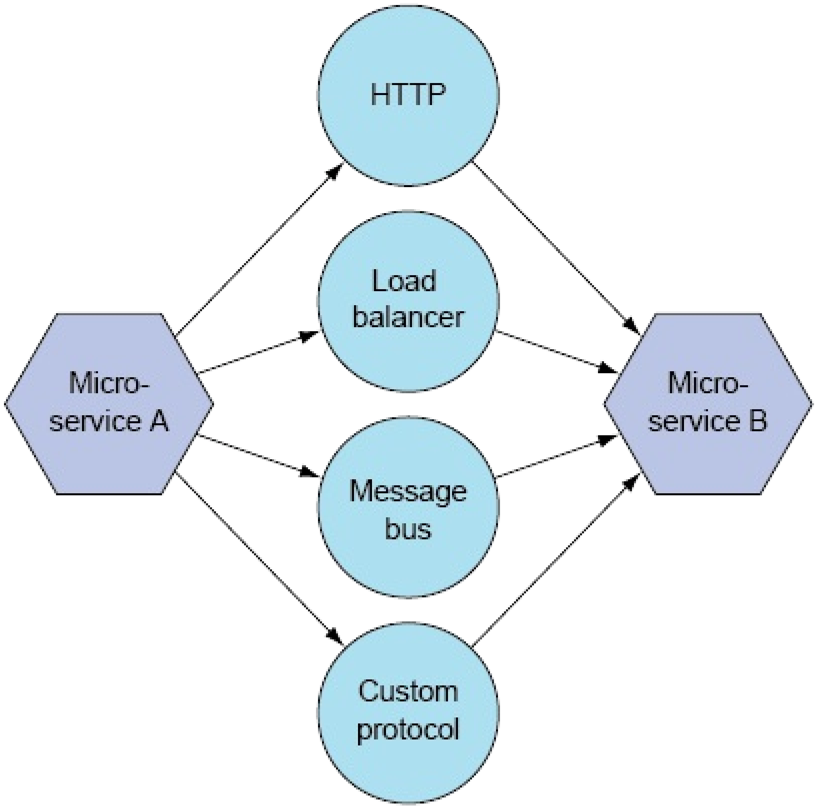
A microservice network can deliver on the axioms by meeting a small set of technical capabilities. These are *transport independence* and *pattern matching*, which together give you *additivity*.

Transport independence

Transport independence is the ability to move messages from one microservice to another without requiring microservices to know about each other or how to send messages. If one microservice needs to know about another microservice and its message protocol in order to send it a message, this is a fatal flaw. It breaks the *no privilege* axiom, because the receiver is privileged from the perspective of the sender. You’re no longer able to compose other microservices over the receiver without also changing the sender.

There are degrees of transport independence. Just as all programming languages are executed ultimately as machine code, all message-transport layers must ultimately resolve senders and receivers to exact network locations. The important factor is how much information is exposed to the internal business logic of the microservice to allow it to send a message. Different message transports (shown in figure 1.6) have different levels of coupling.

Figure 2.1. The underlying infrastructure that moves messages shouldn’t be known to microservices.

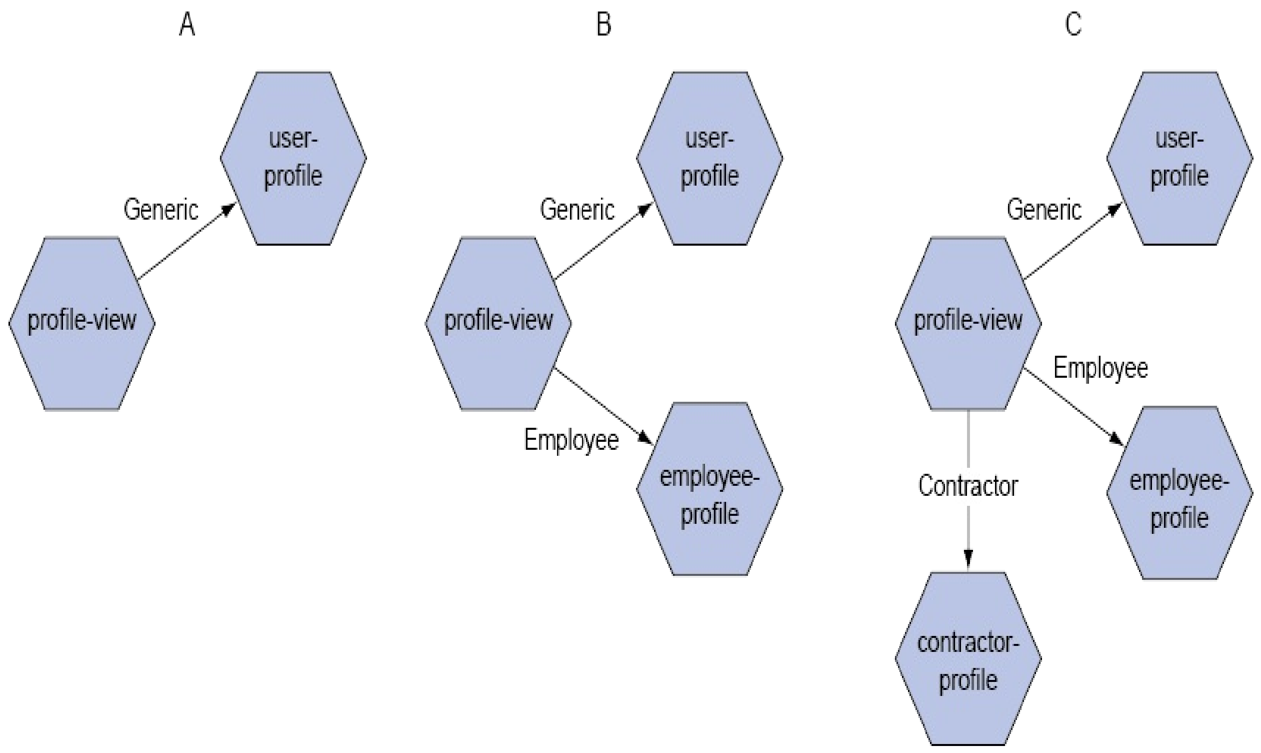


For example, requiring a microservice to look up another service using a service registry is asking for trouble, because doing so creates a dangerous coupling. But service discovery need not be so blatant. You can hide services behind load balancers. The load balancers must know where the services are located, so you haven’t solved the problem; but you’ve made services easier to write, because they only need to find the load balancer, and they’ve become more transport independent. Another approach is to use message queues. Your service must still know the correct topics for messages, and topics are a weak form of network address. The extreme degree, where microservices know nothing of each other, is the goal, because that gives you the full benefits of the architecture.

Pattern matching

Pattern matching is the ability to route messages based on the data inside the messages. [23] This capability lets you dynamically define the network. It allows you to add and remove, on the fly, microservices that handle special cases, and to do so without affecting existing messages or microservices. For example, suppose the initial version of your enterprise system has generic users; but later, new requirements mean you have different kinds of users, with different associated functionalities. Instead of rewriting or extending the generic *user-profile* service (which still does the job perfectly well for generic users), you can create a new *user-profile* service for each kind of user. Then, you can use pattern matching to route user profile request messages to the appropriate service. The power of this approach to reduce technical debt is evident from the fact that no knowledge or dependency exists between different user profile services—they all think that only their kind of user exists. Figure 1.7 shows how the user profile functionality is extended by new microservices.

Figure 2.2. Introducing support for new user profiles

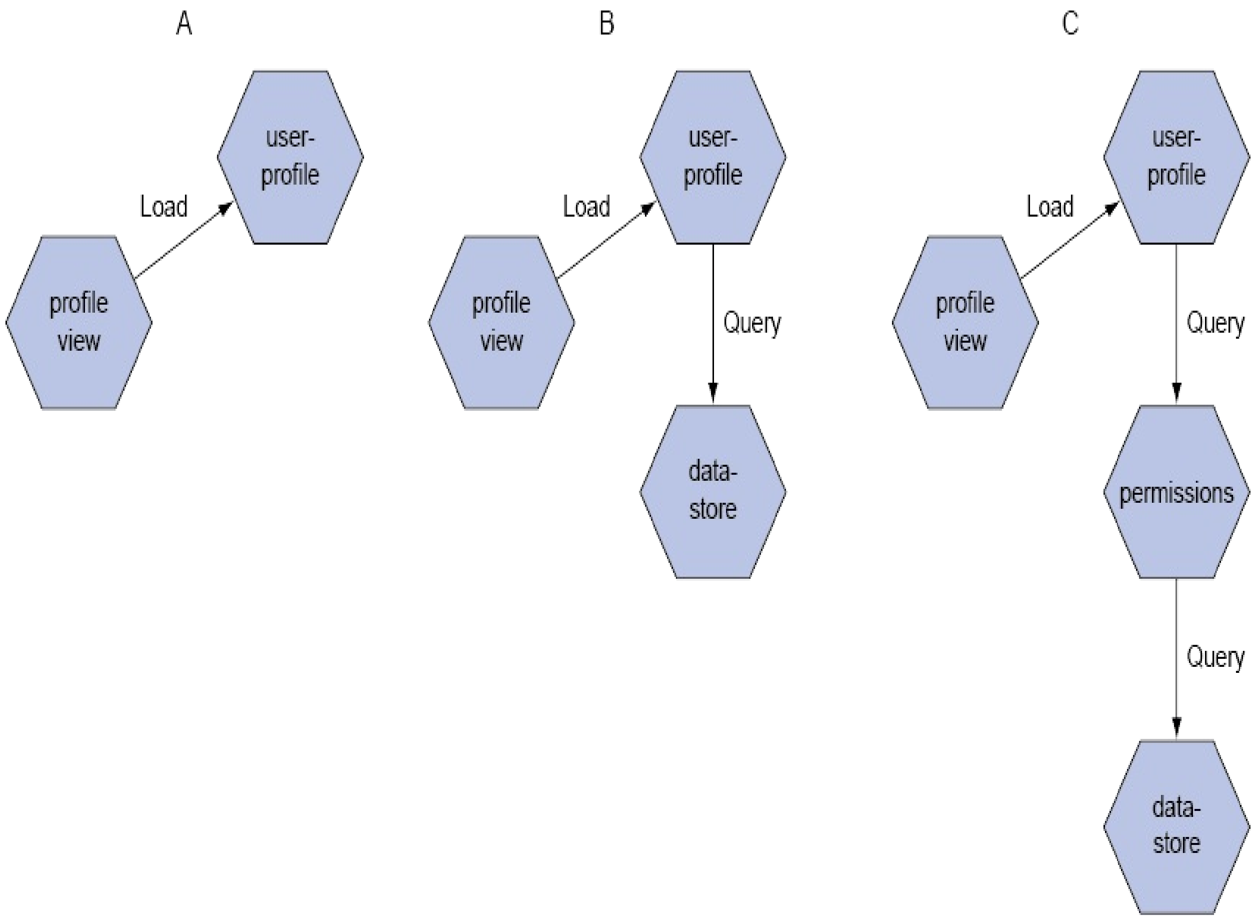


Pattern matching is also subject to degrees of application. The allocation of separate URL endpoints to separate microservices via a load balancer matching against HTTP request paths is an example of a simple case. A full-scale enterprise service bus with myriad complex rules is the extreme end case. Unlike transport independence, the goal isn't to reach an extreme. Rather, you must seek to balance effectiveness with simplicity. You need pattern matching that's deep enough to express business requirements yet simple enough to make composition workable. There's no right answer, and the *uniform communication* axiom's exhortation to keep things simple and homogeneous reflects that. This book proposes that URL matching isn't powerful enough and provides examples of slightly more powerful techniques. There are many ways to shoot yourself in the foot, and it's easy to end up with systems that are impossible to reason about. Err on the side of simplicity, and preserve composability!

Additivity

Additivity is the ability to change a system by adding new parts (see figure 2.3). The essential constraint is that other parts of the system must not change. Systems with this characteristic can deliver very complex functionality and be very complex themselves, and yet maintain low levels of technical debt.footnote: [The venerable Emacs editor is an example of such a system. It’s incredibly easy to extend, despite being a relic of the 1970s. LISP supports additivity.] Technical debt is a measure of how hard it is to add new functionality to a system. The more technical debt, the more effort it takes to add functionality. A system that supports additivity is one that can support ongoing, unpredictable changes to business requirements. A messaging layer based on pattern matching and transport independence makes additivity much easier to achieve because it lets you reorganize services dynamically in production.

Figure 2.3. Additivity allows a system to change in discrete, observable steps.



There are degrees of additivity. A microservice architecture that relies on the business logic of microservices to determine the destination of messages will require changes in both the sender and receiver when you add new functionality. A service registry won’t help you, because you’ll still need to write the code that looks up the new service. Additivity is stronger if you use intelligent load balancers, pattern matching, and dynamic registration of new upstream receivers to shield senders from changes to the set of receivers. Message bus architectures give you the same flexibility, although you'll have to manage the topic namespace carefully. You can achieve near-perfect additivity using peer-to-peer service discovery.[24]. You can even fake additivity by hard-coding service locations inside your message transport code (and updating on deployment), still hiding the services from your business logic.

By supporting the addition of new microservices and allowing them to wrap, extend, and alter messages that are inbound or outbound to other services, you can satisfy the *composition* axiom.

[20] If your microservices communicate only by messages, and you abstract away the transportation of messages, consider how easy it is to work on them locally in a monolith configuration, using function calls as your transport mechanism, and your language's module system to represent services.

[21] Many criticisms of the microservice architecture use the straw man argument that managing and understanding hordes of web server instances exposing HTTP REST APIs hurts too much. Well, if it hurts, stop doing it!

[22] The appropriate pejorative is "Distributed monolith!"

[23] Message routing is free to use any available contextual information to route messages and isn't necessarily limited to data within the messages. For example, microservices under high load can use back-pressure alerts to force traffic onto microservices with less load.

[24] The SWIM algorithm, which provides efficient dissemination of group membership over a distributed network, is an example of the work emerging in this space. See Abhinandan Das, Indranil Gupta, and Ashish Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol," *Proceedings of the International Conference on Dependable Systems and Networks* (2002), http://www.cs.cornell.edu/~asdas/research/dsn02-swim.pdf.

2.4 Practical implications

The practice of enterprise development is riven with *doublethink*. [25] The accumulation of technical debt is the most politically expedient short-term success strategy. The organization simultaneously demands perfect software while imposing constraints that can't be satisfied. Everybody understands the consequences yet participates in what might be called "quality theater," because to do otherwise is a career-limiting move. The expectation of perfection, without understanding its true cost, is the root cause of most software development headaches. [26] Both developers, chasing intellectual highs by building castles in the sky, and the business, lacking an engineering mindset to make trade-offs, are guilty.

Sometimes, this problem can be solved with an edict from on high. Facebook's infamous mantra to "move fast and break things" implicitly acknowledges that perfection is too expensive to justify. Enterprise software developers don't often have the option to use this approach. The more usual business solution is to externalize the problem by converting it into high-stress software development death marches.

The microservice architecture is both a technical and a political strategy. It takes a cannon to intellectual castles and demands honesty from the business. The payoff includes regular work hours and accelerated delivery of business goals. From a technical standpoint, it removes many opportunities for technical debt to accumulate and makes large, distributed teams easier to manage. From a business standpoint, it forces acceptance of system failures and defects. A more honest discussion of the acceptable levels of failure can then begin, and business leaders can make accurate decisions about return on investment in the face of quantified risks. This is something they're good at, given the chance.

2.4.1 Specification

Given that microservices are different from traditional architectures, how do you specify such systems? The first thing to realize is that microservices aren't a radical or revolutionary approach. The architecture emphasizes a component-oriented mindset. As developers, this allows us to ignore language frills. If class hierarchies, object relationships, and data schemas were so effective, we wouldn’t be firing blanks all the time, would we?

Microservice system design is very direct. You take the informal business requirements, [27] determine the behaviors of the system, and map the behaviors to messages. This is a messages-first approach. Ironically, designing a microservice system doesn’t start by asking what services to build; it starts by asking what messages the services will exchange. Once you have the messages, natural groupings will suggest the services to build.

Why do things this way? You have a direct route from business requirements to implementation. This route is traceable and even measurable, because message behavior in production represents desired business outcomes. Your design is independent of implementation, deployment, and data structures. This flexibility allows you to keep up with changing business requirements. Finally, you get a *domain language*—the list of messages is a language that describes the system. From this language, you can derive shared understanding, quality expectations, and performance constraints, and validate correctness in production. All of this is much easier than traditional systems, because messages are homogeneous: they’re always the same kind of thing. [28]

The practice of microservice specification is guided by a single key principle: *move from the general to the specific*. First, you solve the simplest general problem you can. Then, you add more microservices to handle special cases. This approach preserves additivity and delivers the core benefit of microservices: lower technical debt. Let’s say you’re building an internal employee-management system for a global organization. You can start with the simple case of local employees and local regulations. Then, you add microservices for regional community rules, such as those for the United States or the EU. Finally, you handle each country as a special case. As the project progresses, you can deliver more and more value and handle larger and larger subsets of the employee base. Compare this approach to the more common attempt to design sufficiently complex and flexible data structures to handle the entire employee base. Inevitable exceptions will disrupt the integrity of the initial design, ricocheting off false assumptions to cause the collateral damage of technical debt.

2.4.2 Modelling

How do you keep the mapping from requirements to messages to services under control? And how do you manage all the other latent information that need to be well-defined in your system, like configuration, data entity fields, validations, integration settings, and so forth?

In the early years of the microservice architecture, many practioners sought to keep microservices as isolated from each other as possible. This approach has not been as useful as you might think. In practice, systems of microservices do need to share a great deal of latent information. It is necessary to build a shared model of the world. This sometimes called a "Bounded Context", within which you define a "Unbiquitous Language" to avoid ambiguity [29].

I have found that using a formally defined model of the system, expressed in code that can be validated, rather than documents for humans, is a great aid to keeping technical debt at bay. The language and mechanism that defines this model is not that important, it just has to be good enough to give you some confidence that your model is valid and consistent [30].

2.4.3 Deployment

The systems-management needs of microservices are presented as a weakness of the architecture. It’s true that managing the deployment of many hundreds or thousands of individual services in production isn’t a trivial task; but this is a strength, not a weakness, because you’re forced to automate the management of your system in production. Monoliths are amenable to manual deployment, and it’s just about possible to get by. But manual deployment is extremely risky and the cause of a great deal of stress for project teams. [31] The need to automate the deployment of your microservices, and the need to face this requirement early, raises the professionalism of the entire project, because another hiding place for sloppy practices is shot down.

The volume of microservices in production also makes scaling much easier. By design, you’re already running multiple instances of many services, and you can scale by increasing the number of running instances. This scaling ability is more powerful than it first appears, because you can scale at the level of system capabilities. Unlike with monoliths, where scaling is all or nothing, you can easily have variable numbers of different microservices running, applying scaling only where needed. This is far more efficient. There’s no excuse for not doing this, and it isn’t a criticism of the microservice architecture to say that it’s difficult to achieve—this is a baseline feature of any reasonable deployment tool or cloud platform.

The fault tolerance of your overall system also increases. No individual microservice instance is that important to the functioning of the system as a whole. Taking this to an extreme, let’s say you have a high-load microservice that suffers from a memory leak. Any individual instance of this microservice has a useful lifetime of only 15 minutes before it exhausts its memory allocation. In the world of monoliths, this is a catastrophic failure mode. In the world of microservices, you have so many instances that it’s easy to keep your system running without any downtime or service degradation, and you have time to debug the problem in a calm, stress-free manner.[32]

The monitoring of microservice systems is different from that of monoliths, because the usual measurements of low-level health, such as CPU load and memory usage, are far less significant. From a system perspective, what matters is the flow of messages. By monitoring the way messages flow through the system, you can verify that the business requirements and rules are being met, because there’s a direct mapping from behavior to messages. We’ll explore this in detail in chapter 6.

Monitoring message-flow rates is akin to the measurements that doctors use. Blood pressure and heart rate tell a physician much more about a patient that the ion-channel performance of any given cell. Monitoring at this higher level reduces deployment risk. You only ever change the system one microservice instance at a time. Each time, you can verify production health by confirming expected flow rates. With microservices, continuous deployment is the default way to deploy. The irony is that this mode of deployment, despite being far more frequent and subject to less testing, is far less risky than big-bang monolithic deployments, because its impact is so much smaller.

2.4.4 Security

It’s important to step away from the notion that microservices are nothing more than small REST web services. Microservices are independent of transport mechanism, and fixation on the security best practices of any one transport in itself creates exposure. As a guiding principle, external entry points to the system shouldn’t be presented in the same way as internally generated messages.

Microservices that serve external clients should do so explicitly, using the desired communication mechanism of the client directly. The requests of external clients shouldn’t be translated into messages and then presented to the microservice. There’s considerable temptation to do this, because it’s more convenient. A microservice that delivers HTTP content or serves an API endpoint should do so explicitly. Such a microservice may receive and emit internal messages in its interactions with other microservices, but at no time should there be any question of conflation. The microservice’s purpose is to expose an externally facing communication mechanism, which falls outside the communication model between microservices.

It shouldn’t be possible to tell from the outside that a system uses the microservice architecture. Security best practices for microservice systems are no different than for monoliths. The implementation of a hard and secure system boundary is always necessary and is independent of the internal network topology.

The microservice architecture must also deal with the additional exposure created the large numbers of network actors sending and receiving messages. Just because the boundary of the network is secure doesn’t mean communication between microservices doesn’t need to be secured.

Securing messages between services is best handled by a messaging layer that can provide the desired level of security. The messaging layer should handle certificate verification, shared secrets, access control, and other mechanisms of message validation. The need for security at this level casts a harsh light on naïve approaches to microservice communication. Using HTTP utility libraries directly within your microservices means you have to get the security configuration right every time, in each microservice.

Getting the security aspect right is tricky because you’ll have to work against a strict checklist from your enterprise security group. No such group is known for its compromising spirit. This is another advantage of the message abstraction layer—it gives you a place to put all the security implementation details, keeping them away from business logic. It also gives you a way to validate your security infrastructure independently of other layers of the system.

2.4.5 People

The software development process isn’t independent of the architecture under development. If you look closely, you’ll observe that the monolithic architecture drives many behaviors we’ve come to accept as necessary. For example, why are daily stand-ups so important to the agile process? Perhaps it’s because monolithic code bases are highly sensitive to unintended consequences. It’s easy for one developer’s work over here to break another’s over there. Careful branching rituals exist for the same reason. Open source projects that rely heavily on plugin architectures don’t seem to need stand-ups in the same way—perhaps because they use a component model.

The ceremonial demands of software development methodologies are dreamed up with the best of intentions. They’re all attempts to tame complexity and keep technical debt under control. Some methodologies focus on psychology, some on estimation, and others on a rigorous process. They may improve matters, but if any of them had a significant beneficial effect, we’d all adopt that practice without further discussion. [33] The counterargument is that most projects aren’t doing it right, and if only they would be disciplined in their approach, then they would be successful. One response is to observe that a methodology so fragile that most teams can’t execute it correctly is not fit for the purpose.

Unit testing stands out as a practice that has seen wide adoption because it has clear benefits. This is even more of an indictment of other practices in the traditional methodologies of software development. Unfortunately, because it’s so effective, the unit-testing tail often wags the development dog. Projects impose global unit-testing requirements on all code—you must reach such and such a level of coverage, every method must have a test, you must create complex mock objects, and so forth. Stand back and ask these questions:

Does all code need the same coverage? Is all code subject to the same production constraints? Some features deliver orders of magnitude more business value than others—shouldn't you spend more time testing them and less time testing obscure features? The microservice architecture gives you a useful unit of division for quality. Different microservices can have different quality levels as a matter of deliberate choice. This is a more efficient way to allocate resources.

The microservice is also a unit of labor and thus a unit of estimation. Microservices are small, and it's easier to estimate the amount of work required to build small things. This means more-accurate project estimates, especially if you can follow the strategy of moving from general to specific services over time. The older, more general services don't change, so your estimation remains accurate in the later stages of the project, unlike more-traditional code bases where estimation accuracy deteriorates over time as internal coupling increases.

Finally, microservices are disposable. This is perhaps one of the greatest benefits of the architecture from a people perspective. Software developers are known for their large, fragile egos. There's an economic explanation for this: it takes a great deal of effort to build a working monolith and a great deal of further investment to maintain it. This is accidental (rather than essential) knowledge, and it remains valuable only as long as the code base remains in production. From the perspective of a rational economic actor, there's every incentive to keep old spaghetti code alive. Microservices don't suffer from the same effect. They're easy to understand by inspection—there isn't much code. And they get replaced all the time; you write so many of them that individual ones aren't that important. The microservice architecture cleans itself.

[25] A fabulously useful concept from George Orwell's *1984* : "The power of holding two contradictory beliefs in one's mind simultaneously, and accepting both of them."

[26] The cost of the space shuttle software system, one of the most defect-free code bases ever written, has been estimated to be at least $1,000 per line of code. See Charles Fishman, "They Write the Right Stuff," *Fast Company*,

December 31, 1996, http://www.fastcompany.com/28121/they-write-right-stuff. The doublethink of enterprise software development is that space shuttle quality is possible without space shuttle spending.

[27] Business requirements aren’t made formal by accumulating detail. Better to avoid overspecification and stay focused on the desired business outcomes.

[28] *Non sunt multiplicanda entia sine necessitate*, otherwise known as *Occam’s razor*, is the philosophical position that entities must not be multiplied beyond necessity. It puts the onus on those adding complexity to justify doing so. Object-oriented systems provide lots of ways for objects to interact, without much justification.

[29] The original definition of this approach can be found in the book Domain Driven Design, by Eric Evans

[30] Here are some languages and tools to try: YAML, Jsonnet, Pulumi, Cuelang, Dhall, and of course, Nix.

[31] On the morning of Wednesday, August 1, 2012, Knight Capital, a financial services firm on the New York Stock Exchange, lost $460 million in 45 minutes. The firm used a manual deployment process for its automated trading system and had erroneously updated only seven of eight production servers. The resulting operation of the new and old versions of the system, with mismatched configuration files, triggered a trading test sequence to execute in the real market. Automate your production systems! For a detailed technical analysis, see “In the Matter of Knight Capital Americas LLC,” Securities Exchange Act Release No. 34-70694, October 16, 2013, http://www.sec.gov/litigation/admin/2013/34-70694.pdf.

[32] This benefit is best expressed by Adrian Cockcroft, a former director of engineering at Netflix: “You want cattle, not pets.”

[33] To use an example from another domain, the adoption of wheeled luggage by airline passengers was astonishing in the speed of its universal adoption.

2.5 What you get for your money

Microservices can plausibly address the needs of custom enterprise software. By aligning the software architecture of the system more closely with the real intent of the business, software projects can have far more successful outcomes. Real business value is delivered more quickly. Microservice systems approach MVP status faster and thus can be put into production sooner. Once in production, they make it easier to keep up with changing requirements.

Waste and rework, also known as *refactoring*, are reduced because complexity within each microservice can’t grow to dangerous levels. It’s more efficient and easier to write new microservices to handle business change than it is to modify old ones. As a software developer, you can make a bigger professional impact with the evolutionary approach to systems building offered by microservices. It lets you be successful by making the best use of your time. It also frees you from the collective delusion that the building of intricate monolithic code can be accurately estimated. Instead, your value is clear from day one, and discussions with the business are of healthier variety, focused on how you can add value, not on gaming indirect measures of code volume.

In the next chapter, we’ll start to explore the technical foundations of the microservice architecture, starting with the nature of the messages.

2.6 Summary

- The easy accumulation of technical debt is the principle failure of the monolithic architecture.
- Technical debt is caused by language platforms and architectures that make a high degree of coupling possible. Components know too much about each other.
- The most effective way to hide components from each other is to compose them together. Composition allows you to make new components from existing ones, without accelerating the growth of complexity.

- The microservice architecture provides a component model that provides a strong composition mechanism. This enables additivity, the ability to add functionality by adding new parts rather than modifying old ones.
- The practical implications of the microservice architecture demand that you let go of some dearly held beliefs, like uniform code quality, and force acceptance of certain best practices, like deployment automation. These are all good things, because you end up with more efficient resource allocation.

3 Messages

This chapter covers

- Using messages as the key to designing microservice architectures
- Deciding when to go synchronous, and when to go asynchronous
- Pattern matching and transport independence
- Examining patterns for microservice message interactions
- Understanding failure modes of microservice interactions

The term *microservices* invites you to think in terms of services. You'll naturally be drawn to ask, "What are the microservices in this system?" Resist that temptation. Microservice systems are powerful because they allow you to think in terms of messages. If you take a messages-first approach to your system design, you free yourself from premature implementation decisions. The intended behavior of the system can be described in terms of a language of messages, independent of the underlying microservices that generate and react to those messages.

3.1 Messages are first-class citizens

A messages-first approach is more useful than a services-first approach to system design. Messages are a direct expression of intent. Business requirements are expressed as the activities that should happen in the system. We're traditionally taught to extract the nouns from business requirements, so that we can build objects. In practice objects as an abstraction are not found to be a good way to express requirements, because they are conceptually at a level below the requirements.

If you break business requirements down into activities, this naturally suggests the messages within the system. Messages represent actions, not things. Take, for example, a typical e-commerce website. You put some items in your shopping cart, and then you proceed to the checkout. On checkout, the system records the purchase, sends you an email confirmation, and sends the warehouse delivery instructions, among other things. Doesn’t that naturally break down into some obvious activities (see table 3.1.)?

Table 3.1. Messages representing activities

Activity description	Message name	Message data
Checking out	checkout	Cart items and prices
Recording the purchase	record-purchase	Cart items and prices; sales tax and total; customer details
Sending an email confirming the purchase	checkout-email	Recipient; cart summary; template identifier
Delivering the goods	deliver-cart	Cart items; customer address

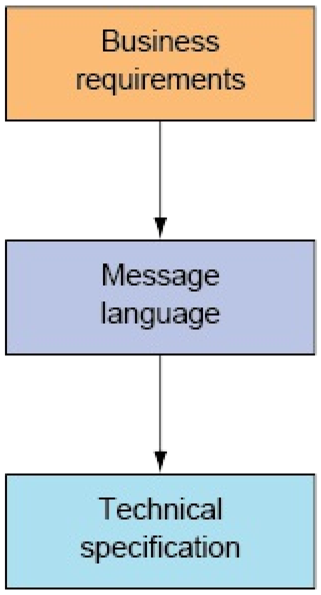
Some of these messages might be: checking out, recording the purchase, sending an email confirming the purchase, delivering the goods. Write down the activities in a table, write down a code name for the activity (which gives you a name for the message), and write down the data contents of the message. By doing this, you can see that there’s no need at this level of analysis to think about which services handle these messages.

Analytical thinking, using messages as a primitive element, can *scale*. No matter how many business activities, and no matter how fine-grained, you can always map them to a message, and think about what data should be contained in that message. Designing the system using services as the primitive element doesn’t scale in the same way. Although there are fewer kinds of services than messages, in practice it’s hard not to think of them from the perspective of network architecture. You have to decide which services talk to which other services, which services handle which messages, and whether services observe or consume messages. You end up having to think about many more different kinds of things. Even worse, you lock yourself into architectural decisions. And your thinking is static from the start, rather than allowing you to adapt message behavior as requirements change and emerge.

Messages are just one kind of thing. They’re homogeneous entities. Thinking about them is easy. You can list them, and you can list the messages that each message generates, so you have a natural description of causality. You can group and regroup messages to organize them conceptually, which may suggest appropriate services to generate or handle those messages. Messages provide a conceptual level that bridges informal business requirements and formal, technical, system specifications. In particular, messages make services and their communication arrangements of less interest—these are implementation details.

The three-step analytical strategy discussed in this book (requirements to messages to services) is shown in figure 3.1.

Figure 3.1. Conceptual levels of system design



3.1.1 Synchronous and asynchronous

Microservice messages fall into two broad types: synchronous and asynchronous. A synchronous message is composed of two parts: a request and a response. The request is the primary aspect of the message, and the response is secondary. Synchronous messages are often implemented as HTTP calls, because the HTTP protocol is such a neat fit to the request/response model. An asynchronous message doesn't have a directly associated response. It's emitted by a service and observed or consumed later by other services.

The essential difference between synchronous and asynchronous messages isn't in the nature of the message transfer protocol; it's in the intent of the originating microservice. The originator of a synchronous message needs an immediate response to continue its work and is blocked until it gets a response. The originator of an asynchronous message isn't blocked, is prepared to wait for results, and can handle scenarios where there are no results:[34]

- *Synchronous*—A *shopping-cart* service, when adding a product to a cart, needs the *sales-tax* service to calculate the updated sales tax before providing a new total to the user. The scenario is a natural fit for a synchronous message.
- *Asynchronous*—Alternatively, to display a list of recommended products below the current shopping cart, the *recommendation* service first issues a message asking for recommendations. Call this a *need* message. Multiple recommendation services, using different algorithms, may respond with messages containing recommendations. The *recommendation* service needs to collect all of these recommendations, so call these *collect* messages. The *recommendation* service can aggregate all the *collect* messages it receives to generate the list of recommendations. It may receive none, in which case it falls back to some default behavior (say, offering a random list of recommendations, after a timeout).

Synchronous and asynchronous messages are convertible

Workflows that use synchronous messages can always be converted to workflows that use asynchronous messages. Any synchronous message can be broken into explicit request and response messages, which are asynchronous. Any asynchronous message that triggers messages in reaction can be aggregated into a single synchronous message that blocks, waiting for the first response.

Beware: Moving between these message workflows requires refactoring of your microservices. The decision to make a given message type synchronous or asynchronous is a core design decision and an expression of your intent as an architect.

3.1.2 When to go synchronous

The synchronous style is well suited to the command pattern, where messages are commands to do something. The response is an acknowledgment that the thing to be done, was done, with such-and-such results. Activities that fit this model include data-storage and data-manipulation operations, instructions to and from external services, serial workflows, control instructions, and, perhaps most commonly, instructions from user interfaces.

The user-interface question

Should user interfaces be built using the microservice architecture? In the first edition of this book, that was an open question, because nobody had seriously tried.

Since then, **micro-frontends** have become an architectural option [35]. I have used the micro-frontend architecture myself. It is most useful when the user interface is built by multiple teams, and delivered from multiple backend microservices.

The very same basic principles can be applied to micro-frontends as those discussed in this book with respect to the backend. However, you will find that most user interface frameworks do not offer the appropriate level of coarse component granularity, dynamic loading, and flexible messaging, and you will need to twist them to your purpose. Since modern user interface development is highly dependent on frameworks, both at the code and cultural level, this implementation friction is not easy to avoid.

Synchronous messages are a naturalistic style and can often be the first design that comes to mind. In many cases, a serial workflow of synchronous messages can be unwound into a parallel set of asynchronous messages. For example, when building a complete content page of a website, each content unit is a mostly independent rectangular area. In a traditional monolith, such pages are often built with simple linear code that blocks, waiting for each item of content to be retrieved. It's more effort to put in place infrastructure code to parallelize the page construction. In a microservice context, a first cut of the page-construction service might work the same way by issuing content-retrieval messages in series, following the linear mental model of the monolith. But because the microservice context offers an asynchronous model as well, and because the page-construction service is isolated from the rest of the system, it's far less effort to rewrite the service to use a parallel approach.[36] Thus, it shouldn't be a matter of much anxiety if you find yourself designing a system that uses synchronous messages to a significant degree.

Business requirements that are expressed as explicit workflows tend to need synchronous messages as part of their implementation. You've already seen such a workflow in the example of the e-commerce checkout process. Such workflows contain gates that prevent further work unless specific operations have completed, and this maps well to the request/response mental model. Traditionally, heavyweight solutions are often used to define such workflows; but in the microservice world, the correct approach is to encode the workflow directly in a special-purpose orchestrating microservice. In practice, this orchestration typically involves both synchronous and asynchronous elements.

Synchronous messages do have drawbacks. They create stronger coupling between services. They can often be seen as remote procedure calls, and adopting this mindset leads to the distributed monolith anti-pattern. A bias toward synchronous messages can lead to deep service dependency trees, where an original inbound message triggers a cascade of multilevel synchronous submessages. This is inherently fragile. Finally, synchronous messages block code execution while waiting for a response. In thread-based language platforms,[37] this can lead to complete failure if all message-handling threads become blocked. In event-based platforms,[38] the problem is less severe but is still a waste of compute resources.

3.1.3 When to go asynchronous

The asynchronous style takes a little getting used to. You have to stop thinking in terms of the programming model of function calls that return results, and start thinking in a more event-driven style. The payoff is much greater flexibility. It’s easier to add new microservices to an asynchronous interaction. The trade-off is that interactions are no longer linear chains of causality.

The asynchronous approach is particularly strong when you need to extend the system to handle new business requirements. By announcing key pieces of information, you allow other microservices to react appropriately without needing any knowledge of those services. Returning to the earlier example, a shopping cart service can announce the fact that a checkout has occurred by publishing a *checkout* message. Microservices that store a record of the purchase, send out confirmation emails, and perform delivery can each listen for this message independently. There’s no need for specific command messages to trigger these activities, nor is there a need for the *shopping-cart* service to know about these other services. This makes it easier to add new activities, such as a microservice for loyalty points, because no changes are required to existing production services.

As a general principle, even when you’re using synchronous messages, you should consider also publishing asynchronous information messages. Such messages announce a new fact about the world that others can choose to act on. This gives you an extremely decoupled, extensible architecture.

The drawback of this approach is the implicit nature of the system’s behavior. As the number of messages and microservices grows, understanding of all the system interactions will be lost. The system will begin to exhibit emergent behavior and may develop undesired modes of behavior. These risks can be mitigated by a strictly incremental approach to system change twinned with meaningful measurement of system behavior. Microservices can’t evaporate inherent complexity, but at least they make it more observable.

3.1.4 Thinking distributed from day one

Microservice systems are distributed systems. Distributed systems present difficult problems that are often intractable. When faced with such challenges, there’s a psychological tendency to pretend they don’t exist and to hope that they’ll just go away by themselves. This style of thinking is the reason many distributed computing frameworks try to make remote and local appear the same: remote procedures and remote objects are presented behind facades that make them look local.[39] This approach trades temporary relief for ongoing insanity. Hiding inherent complexity and the fundamental properties of a system makes the architecture fragile and subject to catastrophic failure modes.

Fallacies of distributed computing

The following fallacies are a warning to programmers everywhere to tread carefully when you take your first steps into the world of distributed computing. They were first outlined informally by David Deutsch (a Sun Microsystems engineer) in 1994:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn’t change.
- There’s one administrator.
- Transport cost is zero.
- The network is homogeneous.

Microservices don’t allow you to escape from these fallacies. It’s a feature of the microservice world view that we embrace the warnings of the fallacies rather than try to solve them.[40]

No matter how good your code is, distributed computing will always be difficult. This is because there are fundamental logical limits on the ability of distributed systems to communicate consistently. A good illustration is the problem of the Byzantine Generals.[41]

Let’s start with two generals of the Byzantine Empire: one is the leader, and the other is a subordinate. Each general commands an army of 1,000 soldiers on opposite sides of an enemy army of 1,500. If they both attack at the same time, their combined force of 2,000 will be victorious. To attack alone will ensure certain defeat. The leader must choose the time of the attack. To communicate, the generals can send messengers who sneak through enemy lines. Some of these messengers will be captured, but the messages are secured with an unbreakable secret code.

The problem is this: what pattern of messages can be used to guarantee that the two generals attack at the same time? A simple protocol suggests itself immediately: require an acknowledgment of each message. But the acknowledgment requires a messenger, who could be captured. And even if the acknowledgment does arrive, the sender doesn’t know this.

Place yourself in the position of the subordinate general. You receive this message: “Attack at dawn!” You acknowledge the message by sending back a messenger of your own. Dawn arrives—do you attack?

What if your acknowledgment messenger was captured? You have no way of knowing if they arrived safely. And your leader has no way of knowing what you know. You’ll both reach the logical conclusion not to attack, even though all messages have been successfully delivered! More acknowledgments won’t help. The last general to send a message can never be sure of its delivery.

For more fun and games, you can increase the number of generals, question the trustworthiness and sanity of both generals and messengers, allow the enemy to subvert or invent messages, and so on. The general case of the Byzantine Generals problem isn’t so far from the reality of running a large-scale microservices deployment!

Pragmatically, this problem is solved by accepting that certainty isn’t possible and then asking how many messages, of what kind, subject to limits such as timeouts, with what meta-information such as sequence numbers, can be used to establish an acceptable level of probability that agreement has been reached. The TCP/IP protocol is a fine example of such a solution.[42]

The key point is that you must accept the unreliable nature of message transmission between microservices. Your design thinking can never be based on an assumption that messages will be delivered as you intend them to be delivered. This assumption isn't safe. Messages aren't like method calls in a monolithic application, where the metaphor of objects exchanging messages is just that: a metaphor. In a distributed system, message delivery can't be made reliable.

What you *can* do is make message delivery—and, indirectly, system behavior—predictable to within acceptable tolerances. You can limit failure by spending money and time. It's your job, and your responsibility as an engineer, to deliver systems to agreed tolerances.

3.1.5 Tactics to limit failure

Just because failure is inevitable doesn't mean you can't do anything about it. As an ethical engineer, you should know and apply reasonable mitigating tactics:

- *Message delivery will be unreliable.* Accept that this is a reality and will happen. You can get closer to 100% reliability by spending ever-larger sums of money, but you'll suffer from diminishing marginal returns, and you'll never get to 100%. Always ask, "What happens if this message is lost?" Timeouts, duplication, logging, and other mitigations can help, but not if you aren't asking that question.
- *Latency and throughput are trade-offs.* Latency tells you how quickly messages are acted on (most commonly, by measuring the response time of synchronous messages). Low latency is a legitimate goal. Throughput tells you how many messages you can handle. High throughput is also a legitimate goal. To a first approximation, these goals are inversely related. Designing a system for high throughput means you need to put in place scalability housekeeping (such as proxy servers) that increases latency, because there are extra network hops. Conversely, designing for low latency means high throughput, although possible, will be much

more expensive (for example, you might be using more-powerful machines).

- *Bandwidth matters.* The networked nature of microservice systems means they're vulnerable to bandwidth limitations. Even if you start out with a plentiful supply, you must adopt a mentality of scarcity. Misbehaving microservices can easily cause an internally generated denial-of-service attack. Keep your messages small and lean. Don't use them to send much data; send references to bulk data storage, instead.[43] Bandwidth as a resource is becoming cheaper and more plentiful, but we aren't yet in a post-scarcity world.
- *Security doesn't stop at the firewall.* You may be tempted to engage in security theater, such as encrypting all interservice messages. Or you might be forced into it by your clients. In some ways, this is mostly harmless, although it does drain resources. It's more effective to adopt the stance, within each microservice, that inbound messages are potentially malign and may come from malicious actors. Semantic attacks[44] are your primary concern. Microservices are naturally more resistant to syntactic attacks because they offer a significantly reduced attack surface—you can only get in via messages. Semantic attacks, in the form of malicious messages generated by improper external interactions, are the principle attack vector. Message schema validation won't help here, because the dangerous messages are exactly those that work and are, by definition, syntactically correct.
- *Avoid local solutions to global problems.* Let's say you've decided to use a synchronous approach to microservices, with HTTP REST as the message transport. Every microservice needs to know the network location of every other microservice it wants to talk to, because it has to direct those HTTP requests somewhere. So what do you do? *Verschlimmbessern*![45] Obviously, a distributed key store running some sort of distributed shared-state consensus algorithm will give you a service-discovery solution, and problem solved! It isn't that this is not a legitimate solution; the real question is whether the problem (service discovery) is one you should be solving in the first place.
- *Automate or die.* There's no such thing as a free lunch. For all the benefits of microservices, you can't escape the fact that you have to manage lots of little things spread over many servers. Traditional tools and approaches won't work, because they can't handle the ramp-up in

complexity. You’ll need to use automation that scales. More on this in chapter 8.

[34] Microservices don’t need to wait on human time scales. The waiting period could be on the order of milliseconds.

[35] And have their very own book: Micro Frontends in Action, Michael Geers, Manning 2020.

[36] This is a good example of the ease of refactoring when you use microservices; a similar refactoring inside a monolith would involve much code munging (to use the technical term). In the microservice case, refactoring is mostly reconfiguration of message interactions.

[37] Such as the Java JVM or .NET CLR.

[38] Node.js is the prime example.

[39] This is a feature of many APIs, such as the one for gRPC, and older approaches such as Java RMI and CORBA.

[40] For a wonderfully sardonic discussion of the fallacies, you won’t do better than Arnon Rotem-Gal-Oz’s paper “Fallacies of Distributed Computing Explained,” http://www.rgoarchitects.com/Files/fallacies.pdf.

[41] Leslie Lamport, Robert Shostak, and Marshall Pease, “The Byzantine Generals Problem,” ACM Transactions on Programming Languages and Systems 4, no. 3 (July 1982), http://www.cs.cornell.edu/courses/cs614/2004sp/papers/lsp82.pdf. This seminal paper explains the basis of distributed consensus.

[42] Transmission Control Protocol/Internet Protocol (TCP/IP) uses algorithms such as slow start (increasing data volume until maximum capacity is found) and exponential backoff (waiting longer and longer when retrying sends) to achieve an acceptable level of transmission reliability.

[43] To send an image between services, don’t send the image binary data, send a URL pointing to the image.

[44] Explained in detail by Bruce Schneier, a respected security expert, on his blog: “Semantic Attacks: The Third Wave of Network Attacks,” Schneier on Security, October 15, 2000, http://mng.bz/Ai4Q.

[45] One of those wonderful German loan words, perhaps even better than schadenfreude. To verschlimmbessern is to make something much worse by earnestly trying to make it better, all the while blissfully ignoring the real problem.

3.2 Case study: Sales tax rules

Our case study in this chapter focuses on one part of the work of building a general e-commerce solution: calculating sales tax. This may not seem very challenging, but a full solution has wonderful hidden depths.

To set the scene, imagine a successful company that has grown quickly and offers products in multiple categories and geographies. The company started out selling one type of good in one country; but then things took off, and the company quickly had to adapt to rapid growth. For the sake of argument, let’s imagine this company began by selling books online and then branched into electronics, shoes, and online services.[46]

Here’s the business problem when it comes to sales tax: you have to account for differences based on many variables. For example, determining the correct rate might involve the category of good, the country of the seller, the country of the buyer, the location in the country of either, local bylaws, the date and time of the purchase, and so on. You need a way to take in all of these variables and generate the right result.

3.2.1 The wider context

An e-commerce website is a good example of a common problem space for microservices. There’s a user interface that must be low latency, and there’s a backend that has a set of horizontal and vertical functionalities. *Horizontal functionalities*, such as user account management, transactional email workflows, and data storage, are mostly similar for many different kinds of applications. *Vertical functionalities* are particular to the business problem at hand; in the e-commerce case, these are functionalities such as the shopping cart, order fulfillment, and the product catalog.

Horizontal functionalities can often be implemented using a standard set of prebuilt microservices. Over time, you should invest in developing such a set of functionalities, because they can be used in many applications, thus kick-starting your development. While working on this project, you'll almost certainly extend, enhance, and obsolesce these starter microservices, because they won't be sufficient for full delivery. How to do this in a safe way, with pattern matching on messages, is one of the core techniques we'll explore in this chapter.

Vertical microservices start as greenfield implementations, which makes them even more vulnerable to obsolescence. You'll invariably make incorrect assumptions about the business domain. You'll also misunderstand the depth of the requirements. The requirements will change in any case, because business stakeholders develop a deeper understanding too. To deal with this, you can use the same strategic approach as with horizontals: pattern matching. And for verticals, it's a vital strategy to avoid running up technical debt.

[46] Our example is deliberately implausible. Such a thing would never catch on.

3.3 Pattern matching

Pattern matching is one of the key strategies for building scalable microservice architectures—not just technically scalable, but also psychologically scalable, so that human software developers can understand the system. A large part of the complexity of microservice systems comes from the question of how to route messages. When a microservice emits a message, where should it go? And when you're designing the system, how should you specify these routes?

The traditional way to describe network relationships, by defining the dependencies and data flows between services, doesn't work well for microservices. There are too many services and too many messages. The solution is to turn the problem on its head. Start instead from the properties of the messages, and allow the network architecture to emerge dynamically from there.

In the e-commerce example, some messages will interact with the shopping cart. There’s a subset of messages in the message language that the shopping cart would like to receive: say, *add-product*, *remove-product*, and *checkout*. Ultimately, all messages are collections of data. You can think of them as collections of key-value pairs. Imagine an all-seeing messaging deity that observes all messages and, by identifying the key-value pairs, sends the messages to the correct service.

Message routing can be reduced to the following simple steps:

1. Represent the message as key-value pairs (regardless of the actual message data).
2. Map key-value pairs to microservices, using the simplest algorithm you can think of.

The way you represent the message as key-value pairs isn’t important. Key-value pairs are just the lowest-common-denominator data representation that has enough information to be useful. The algorithm isn’t important; but as a practical matter, it should be easy enough for fallible humans to run in their heads [47].

This approach—routing based on message properties—has been used many ways; it isn’t a new idea. Any form of intelligent proxying is an example. What’s new is the explicit decision to use it as the common basis for message routing. This means you have a homogeneous solution to the problem and no special cases. Understanding the system reduces to reviewing the list of messages and the patterns used to route them; this can even be done in isolation from the services that will receive them!

The big benefit of this approach is that you no longer need to expose metainformation to the developer sending a message; you don’t need a service address. Service addresses come in many flavors—some not so obvious. A domain name is obviously an address, but so is a REST URL. The topic or channel name on a message bus and the location of the bus on the network are also addresses. The port number representing a remote service, exposed by an overlay network, is still an address! Microservices shouldn't know about other microservices.

To be clear, addressing information has to exist somewhere. It exists in the configuration of the abstraction layer that you use to send and receive messages. Although this requires work to configure, it isn't the same as having that information embedded in the service. From a microservice perspective, messages arrive and depart, and they can be any messages. If the service knows how to deal with a message, because it recognizes that message in some way, all to the good; but it's that microservice's business alone. The mapping from patterns to delivery routes is an implementation detail.

Let's explore the consequences of this approach. You'll see how it makes it much easier to focus on solving the business problem, rather than obsessing about accidental technical details.

3.3.1 Sales tax: starting simple

Let's start with the business requirement for handling sales tax. We'll focus narrowly on the requirement to recalculate sales tax after an item is added to the cart.

When a user adds a product to their shopping cart, they should see an updated cart listing all the products they previously added together with the new one. The cart should also have an entry for total sales tax due.[48] Let's use a synchronous *add-product* message that responds with an updated cart and a synchronous *calculate-sales-tax* message that responds with the gross price, to model this business requirement. We won't concern ourselves with the underlying services.

The list of messages has only two entries:

- *add-product*—Triggers *calculate-sales-tax*, and contains details of the product and cart
- *calculate-sales-tax*—Triggers nothing, and contains the net price and

relevant details of the product

Let’s focus on the *calculate-sales-tax*—what properties might it have?

- Net price
- Product category
- Customer location
- Time of purchase
- Others you haven’t thought of

The microservice architecture allows you to put this question to one side. You don’t need to think about it much, because you’re not trying to solve everything at once. The best practice is to build the simplest possible implementation you can think of: solve the simple, general case first.

Let’s make some simplifying assumptions. You’re selling one type of product in one country, and this fully defines the sales tax rate to apply. You can handle the *calculate-sales-tax* message by writing a *sales-tax* microservice that responds synchronously. It applies a fixed rate, hardcoded into the service, to the net price, to generate a gross price: `gross = net * rate`.

Let’s also return to the useful fiction that every microservice sees every message. How does the *sales-tax* microservice recognize *calculate-sales-tax* messages? Which properties are relevant? In fact, there’s nothing special about the properties listed previously. There isn’t enough information to distinguish this message from others that also contain product details, such as *add-product* messages. The simple answer is to label the message. This isn’t a trick question: labels are a valid way to namespace the set of messages. Let’s give *calculate-sales-tax* messages a label with the string value `"sales-tax"`[49].

The label allows you to perform pattern matching. All messages that match the pattern `label:sales-tax` go to the *sales-tax* microservice. You’re careful not to say how that happens in terms of data flowing over the network; nor are you saying where that intelligence lies. You’re only concerned with defining a pattern that can pick out messages you care about.

Here’s an example message:

```
label: sales-tax
net: 1.00
```

The *sales-tax* service, with a hardcoded rate of, say, 10%, responds like this:

```
gross: 1.10
```

What is the pattern-matching algorithm? There's no best choice. The best practice is to keep it simple. In this example, you might say, "Match the value of the `label` property." As straightforward as that.

3.3.2 Sales tax: handling categories

Different categories of products can have different sales tax rates. Many countries have special, reduced sales tax rates that apply only to certain products or only in certain situations. Luckily, you include a category property in your messages. If you didn't, you'd have to add one anyway. In general, if you're missing information in your messages, you update the microservice generating those messages first, so that the new information is present even if nobody is using it. Of course, if you use strict message schemas, this is much more difficult. It's precisely this type of flexibility that microservices enable; by having a general rule that you ignore new properties you don't understand, the system can continue functioning.

To support different rates for different product categories, a simple approach is to modify the existing *sales-tax* microservice so that it has a lookup table for the different rates. You keep the pattern matching the same. Now you can deploy the new version of *sales-tax* in a systematic fashion, ensuring a smooth transition from the old rules to the new. Depending on your use case, you might tolerate some discrepancies during the transition, or you might use feature flags in your messages to trigger the new rules once everything is in place.

An alternative approach is to write a new *sales-tax* microservice for each category. This is a better approach in the long term. With the right automation and management in place, the marginal cost of adding new microservices is low. Initially, they'll be simple—effectively, just duplicates of the single-rate service with a different hardcoded rate.

You might feel uneasy about this suggestion. Surely these are now *nanoservices* [50]; it feels like the granularity is too fine. Isn't a lookup table of categories and rates sufficient? Perhaps, but the lookup table is an open door to technical debt. It's a data structure that models the world and must be maintained. As new complexities arise, it will need to be extended and modified. The alternative—using separate microservices—keeps the code simple and linear.

The statement of the business problem was misleading. Yes, different product categories have different sales tax rates. But if you look at the details, every tax code of every country in the world contains a morass of subdivisions, special cases, and exclusions, and legislatures add more every day. It's impossible to know what business rules you'll need to apply using your lookup table data model. And after a few iterations, you'll be stuck with that model, because the rest of your system assumes it exists.

Separating the product categories into separate microservices is a good move when faced with this type of business rule instability. At first, it seems like overkill, and you're definitely breaking the DRY (Don't Repeat Yourself) principle, but it quickly pays off. The code in each microservice is more concrete, and the algorithmic complexity is lower.

With this approach, the next question is how to route messages to the new microservices. Pattern matching again comes into play. Let's use a better, but still simple, algorithm. Each sales-tax microservice examines inbound messages and responds to them if they have a `label:sales-tax` property and a category property whose value matches the category they can handle. For example, the message

```
label: sales-tax
net: 1.00
category: standard
```

is handled by the existing *sales-tax* microservice. But the message

```
label: sales-tax
net: 1.00
category: reduced
```

is handled by the *sales-tax-reduced* microservice. The mapping between patterns and services is listed table 3.2.

Table 3.2. Pattern to service mapping

Pattern	Microservice
`label:sales-tax`	sales-tax
`label:sales-tax,category:standard`	sales-tax
`label:sales-tax,category:reduced`	sales-tax-reduced

Notice that the general-case microservice, *sales-tax*, handles messages that have no category. The calculation may be incorrect, but you'll get something back, rather than failure. If you're going for availability rather than consistency, this is an acceptable trade-off.

It's the responsibility of the underlying microservice messaging implementation to adhere to these pattern-matching rules. Perhaps you'll hard-code them into a thin layer over a message queue API. Or maybe you'll use the patterns to construct message-queue topics. Or you can use a discovery service that responds with a URL to call for any given message. For now, the key idea is this: choose a simple pattern-matching algorithm, and use it to declaratively define which types of messages should be handled by which types of services. You're thus communicating your intent as architect of the microservice system.

The pattern-matching algorithm

You might ask, "Which pattern-matching algorithm should I use? How complex should it be?" The answer is, as simple as possible. You should be able to scan the list of pattern-to-microservice mappings and manually assign any given message to a microservice based on the content of the message.

There’s no magic pattern-matching algorithm that suits all cases. The Seneca framework[51] used for the case study in chapter 9 uses the following algorithm:

- Each pattern picks out a finite set of top-level properties by name.
- The value of each property is considered to be a character string.
- A message matches a pattern if all property values match under string equality.
- Patterns with more properties have precedence over patterns with fewer.

Patterns with the same number of properties use alphabetical precedence Given the following pattern-to-microservice mapping:

- `a:0` maps to microservice A.
- `b:1` maps to microservice B.
- `a:0,b:1` maps to microservice A.
- `a:0,c:2` maps to microservice C.

The following messages will be mapped as indicated:

- Message `{a:0, x:9}` goes to A.
    - Matches pattern `a:0`, because property `a` in the message has value 0.
- Message `{b:1, x:8}` goes to B.
    - Matches pattern `b:1`, because property `b` in the message has value 1.
- Message `{a:0, b:1, x:7}` goes to A.
    - Matches pattern `a:0,b:1` rather than `a:0` or `b:1`, because `a:0,b:1` has more properties.
- Message `{a:1, b:1, x:6}` goes to B.
    - Matches pattern `b:1`, because property `b` in the message has value 1, and there’s no pattern for `a:1`.

- Message `{a:0, c:2, x:5}` goes to C.
    - Matches pattern `a:0,c:2` by the values of properties `a` and `c`.
- Message `{a:0, b:1, c:2, x:9}` goes to A.
    - Matches pattern `a:0,b:1` by the values of properties `a` and `b` but not `c`, because `b` is before `c` alphabetically.

In all of these cases, property `x` is data and isn’t used for pattern matching. That said, there’s nothing to prevent future patterns from using `x`, should it become necessary due to emerging requirements.

3.3.3 Sales tax: going global

The business grows, as businesses do. This creates the proverbial nice problem to have: the business wants to expand into international markets. The e-commerce system now needs to handle sales tax in a wide range of countries. How do you do this? It’s a classic case of combinatorial explosion, in that you have to think in terms of per-country, per-category rules.

The traditional strategy is to refactor the data models and try to accommodate the complexity in the models. It’s an empirical observation that this leads to technical debt; the traditional approach leads to traditional problems. Alternatively, if you follow the microservice approach, you end up with per-country, per-category microservices. Doesn’t the number of microservices increase exponentially as you add parameters?

In reality, there’s a natural limit on this combinatorial explosion: the natural complexity size of an individual microservice. As we will see in chapter 4, this limit is about one week of developer effort (per version), perhaps coarsely graded by skill level. This fact means you’ll tend to build data models up to this level of complexity, but no further. And many of these data models can handle multiple combinations of messages.

Certain countries have similar systems with similar business rules. These can be handled by a microservice with that dreaded lookup table. But exceptions and special cases won’t be handled by extending the model: instead, you’ll write a special-case microservice.

Let’s look at an example. Most European Union countries use the standard and reduced-rates structure. So, let’s build an *eu-vat* microservice that uses a lookup table keyed by country and category. If this seems heretical, given the existing *sales-tax* and *sales-tax-reduced* microservices, you could be right—you may need to embrace the heresy, end-of-life those services, and use a different approach. This isn’t a problem! You can run both models at the same time and use pattern matching to route messages appropriately.

3.3.4 Business requirements change, by definition

Any approach to software development that expects business requirements to remain constant over the course of a project is doomed to deliver weak results. Even projects that have strict, contract-defined specifications suffer from requirement drift. Words on paper are always subject to reinterpretation, so you can take for granted that the initial business requirements will change almost as soon as the ink is dry. They will also change during the project and after the project goes live.

The agile family of software project management methodologies attempts to deal with this reality by using a set of working practices, that encourages flexibility. The agile working practices—iterations, unit testing, pair programming, refactoring, and so on—are useful, but they can’t overcome the weight of technical debt that accumulates in monolithic systems. Unit testing, in particular, is meant to enable refactoring, but in practice it doesn’t achieve this effectively.

Why is refactoring a monolithic code base difficult? Because monolithic code bases enable the growth of complex data structures. The ease of access to internal representations of the data model means it’s easy to extend and enhance. Even if the poor project architect initially defines strict API boundaries, there’s no strong defense against sharing data and structures. In the heat of battle, rules are broken to overcome project deadlines.

The microservice architecture is far less vulnerable to this effect, because it’s much harder for one microservice to reach inside another and interfere with its data structures. Messages naturally tend to be small. Large messages aren’t efficient when transported over the network. A natural force makes messages concise, including only the data that’s needed, and in the form of simpler data structures.

Combined, the small size of microservices, the smaller size of messages, and the resulting lower complexity of data structures internal to microservices result in an architecture that’s easier to refactor. This ease derives from the engineering approach, rather than project management discipline, and so is robust in response to variances in team size, ability, and politics.

3.3.5 Pattern matching lowers the cost of refactoring

The pattern-matching tactic has a key role to play in making refactoring easier. When first building microservices, developers have a tendency to apply schema validation to the messages. This is an attempt to enforce a quality of correctness in the system, but it’s misguided.

Consider the scenario where you want to upgrade a particular microservice. For scale, you might have multiple instances of the microservice running. Let’s say this microservice is at version 1.0. You need to deploy version 2.0, which adds new functionality. The new functionality uses new fields in the messages that the microservice sends and receives.

Taking advantage of the deployment flexibility of microservices, you deploy a new instance of v2.0 while leaving the existing v1.0 instances running. This lets you monitor the system to see whether you’ve broken anything. Unfortunately, you have! The strict schema you’re enforcing for v1.0 messages isn’t compatible with the changes in v2.0, and you can’t run both instances of the service at the same time. This negates one of the main reasons for using microservices in the first place.

Alternatively, you can use pattern matching. With pattern matching, as you’ve seen in the sales tax example, it’s easy to modify the messages without breaking anything. Older microservices ignore new fields that they don’t understand. Newer microservices can claim messages with new fields and operate on them correctly. It becomes much easier to refactor.

Postel's law

Jon Postel, who was instrumental in the design of the TCP/IP protocol (among others), espoused this principle: "Be conservative in what you do, be liberal in what you accept from others." This principle informed the design of TCP/IP. Formally, it's defined as being contravariant on input (that is, you accept supersets of the protocol specification) and covariant on output (you emit subsets of the protocol specification). This approach is a powerful way to ensure that many independent systems can continue to work well with one another. It's useful in the microservice world.

This style of refactoring has a natural safety feature. The production system is changed by adding and subtracting microservice instances, but it isn't changed by modifying code and redeploying. It's easier to control and measure possible side effects at the granularity level of the microservice—all changes occur at the same level, so every change can be monitored the same way. We'll discuss techniques for measuring microservice systems in chapter 9—in particular, how measurement can be used to lower deployment risk in a quantifiable way.

[47] Or you can delegate the entire problem to your cloud platform, using something like AWS EventBridge.

[48] For our purposes, consider value-added tax (VAT, as used in Europe) to be calculated in the same way.

[49] You might argue that this label is in fact a service address. It is a very weak address. But there is no hard-coupling to a specific service, and no need to change any code or message schemas if you decide to refactor your services.

[50] A somewhat derisory term for microservices that are just a few lines of code.

[51] I'm the maintainer of this open source project. See http://senecajs.org.

3.4 Transport independence

Microservices must communicate with each other, but this doesn’t mean they need to *know* about each other. When a microservice knows about another microservice, this creates an explicit coupling. One of the strongest criticisms of the microservice architecture is that it’s a more complex, less manageable version of traditional monolithic architectures. In this criticism, messages over the network are nothing more than elaborate remote procedure calls, and the system is a mess of dependencies—the so-called *distributed monolith*.

This problem arises if you consider the idea of service identity to be essential. The naïve model of microservice communication is one where you need to know the identity of the microservice to which you want to send a message: you need the address of the receiving microservice. An architecture that uses direct HTTP calls for request/response messages suffers from this problem—you need a URL endpoint to construct your message-sending calls. Under this model, a message consists not only of the content of the message but also the identity of the recipient.

There’s an alternative. Each microservice views the world as a universe from which it receives messages and to which it emits messages. But it has no knowledge of the senders or receivers. How is this possible? How do you ensure that messages go to the right place? This knowledge must exist somewhere. It does: in the configuration of the transport system for microservice messages. But ultimately, that’s merely an implementation detail. The key idea is that microservices don’t need to know about each other, how messages are transported from one microservice to another, or how many other microservices see the message. This is the idea of *transport independence*, and using it means your microservices can remain fully decoupled from each other.

3.4.1 A useful fiction: the omnipotent observer

Transport independence means you can defer consideration of practical networking questions. This is extremely powerful from the perspective of the microservice developer. It enables the false, but highly useful, assumption that any microservice can receive and send any message. This is useful because it allows you to separate the question of how messages are transported from the behavior of microservices.

With transport independence, you can map messages from the message language to microservices in a completely flexible way. You’re free to create new microservices that group messages in new ways, without impacting the design or implementation of other microservices.

This is why you were able to work at the design level with the messages describing the e-commerce system, without first determining what services to build. Implicitly, you assumed that any service can see any messages, so you were free to assign messages to services at a late stage in the design process. You get even more confidence that you haven’t painted yourself into a corner when you realize that this implicit assumption remains useful for production systems. Microservices are disposable, so reassigning messages to new services is low cost.

This magical routing of messages is enabled by dropping the idea that individual microservices have identities, and by using pattern matching to define the mapping from message to microservice. You can then fully describe the design of the system by listing the patterns that each microservice recognizes and emits. And as you’ve seen with the sales tax example, this is the place you want to be.

The implementation and physical transport of messages isn’t something to neglect—it’s a real engineering problem that any system, particularly large systems, must deal with. But microservices should be written *as if* transport is an independent consideration. This means you’re free to use any transport mechanism, from HTTP to messages queues, and free to change the transport mechanism at any time.

You also get the freedom to change the way messages are distributed. Messages may be participants in request/response patterns, publish/subscribe patterns, actor patterns, or any other variant, without reference to the microservices. Microservices neither know nor care who else interacts with messages.

In the real world, at the deployment level, you must care. We’ll discuss mechanisms for implementing transport independence in chapter 8.

3.5 Message patterns

The core principles of pattern matching and transport independence allow you to define a set of message patterns, somewhat akin to object-oriented design patterns. The are two principle aspects of message interactions that we will use to define these patterns>

This first aspect is the separation between **synchronous** and **asynchronous** messages.

The second aspect is the separation between messages that are **consumed**, and those that are only **observed**. A consumed message is accepted by a one service, and that’s it. No other service will get to see that message. In contrast, an observed message can be seen by more than one service, and no individual service can "remove" the message.

These aspects are orthogonal to each other, so there are four possibilities for a given message:

- synchronous/consumed (e.g. traditional HTTP)
- synchronous/observed (e.g. auditing logs)
- asynchronous/consumed (e.g. message queues)
- asynchronous/observed (e.g. network broadcasts)

We’ll consider these interactions in the context of increasing numbers of services. In all cases, unless explicitly noted, we assume a scalable system where each microservice is run as multiple instances. We assume that the deployment infrastructure, or microservice framework, provides capabilities such as load balancing to make this possible.

To formalize the categorization of the design patterns, you can think of them in terms of the number of message patterns and the number of microservices (not instances!). In the simplest case, there’s a single message of a particular pattern between two microservices. This is a 1/2 design pattern, using the form *m*/*n*, where *m* is the number of message patterns and *n* is the number of microservices.

In the diagrams that follow, we’ll also use some conventions to indicate what kind of message is sent between services:

- *Synchronous/asynchronous* (solid/dashed line)—The message expects/doesn't expect a response.
- *Observe/consume* (empty/full arrowhead)—The message is either observed (others can see it too) or consumed (others can't see it).

We'll establish a full formal style of diagrams for microservices in the next chapter when we move on to services themselves. For now, we have enough to describe messages.

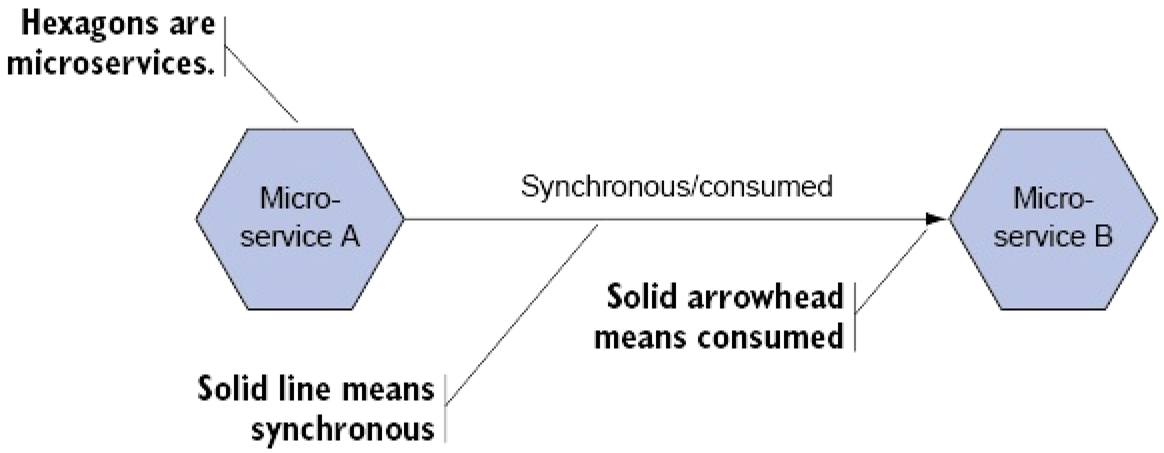
3.5.1 Core patterns: one message/two services

In these patterns, you enumerate the four permutations of the synchronous/asynchronous and observe/consume axes. In general, enumerating the permutations of message interactions, especially with higher numbers of microservices, is a great way to discover possibilities for microservice interaction patterns. But we'll start simple, with the big four.

1/2: Request/response

This pattern, illustrated in figure 3.2, describes the common HTTP message transport model. Messages are synchronous: the initiating microservice expects a response. The listening microservice consumes the message, and nobody else gets to see it. This mode of interaction covers traditional REST-style APIs and a great many first-generation microservice architectures. Considerable tooling is available to make this interaction highly robust.[52]

Figure 3.2. Request/response pattern: the calculation of sales tax, where microservice *A* is the shopping-cart service, microservice B is the sales-tax service, and the message is calculate-sales-tax. The shopping-cart service expects an immediate answer so it can update the cart total.


Hexagons are microservices.
Micro-service A
Synchronous/consumed
Micro-service B
Solid arrowhead means consumed
Solid line means synchronous

If the cardinality of the listening microservice is greater than a single instance, then you have an actor-style pattern. Each listener responds in turn, with a load balancer[53] distributing work according to some desired algorithm.

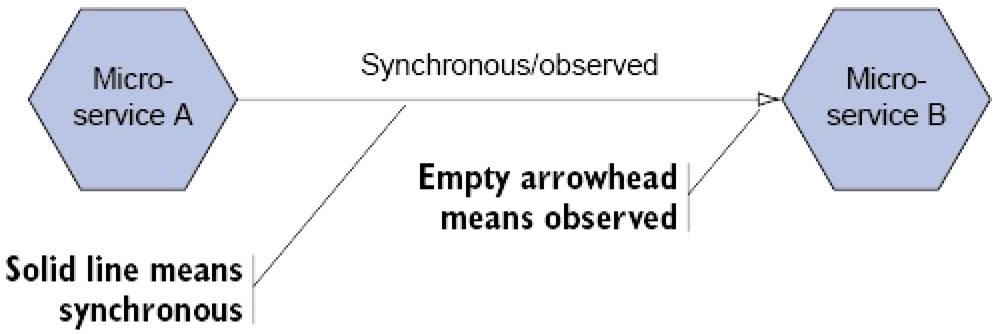
When you develop a microservice system locally, it's often convenient to run the system as a set of individual microservices, with a single instance of each type. Using the power of transport independence, you can run your entire system as a single process, loading each microservice as a pure component [54]. Message transport reduces to function calls. Some services might still need to run as separate processes, in which case you can hardcode some URLs and use HTTP for message transport.

It can't be stressed forcefully enough that *the microservices must be sheltered from the details of this local configuration*. The benefit is that it's much easier to develop and verify message interactions in this simplified configuration.

1/2: Sidewinder

This pattern may at first seem rather strange (see figure 3.3). It's a synchronous message that isn't consumed. So who else observes the message? This is a good example of the level that this model operates at—you aren't concerned with the details of the network traffic. How others observe this message is a secondary consideration. This pattern communicates *intent*. The message is observable. Other microservices, such as an auditing service, may have an interest in the message, but there’s still one microservice that will supply the response. This is also a good example of the way a model generates new ways of looking at systems by enumerating the combinations of elements the model provides.

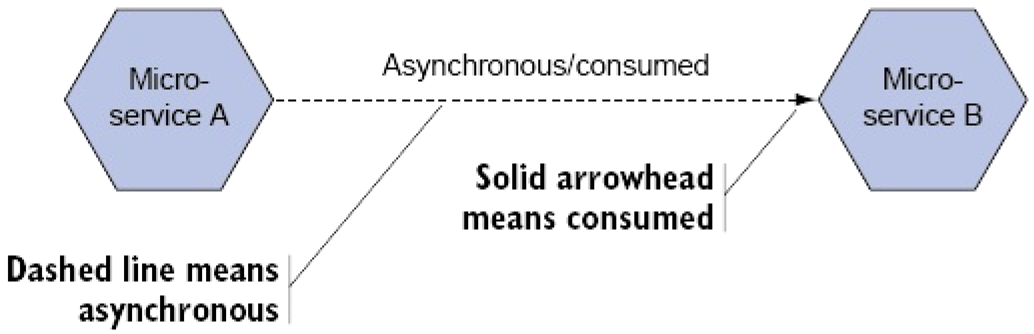
Figure 3.3. Sidewinder pattern: in the e-commerce system, the shopping-cart service may send a checkout message to the delivery service. A recommendation service can observe the checkout messages to understand the purchasing behavior of customers and generate recommendations for other products they might be interested in. But the existence of the recommendation service isn’t known to the shopping-cart and delivery services, which believe they’re participating in a simple request/response interaction.



1/2: Winner-take-all

This is a classic distributed-systems pattern (see figure 3.4). Workers take tasks from a queue and operate on them in parallel. In this world view, asynchronous messages are sent out to instances of listening microservices; one of the services is the winner and acts on the message. The message is asynchronous, so no response is expected.

Figure 3.4. Winner-take-all pattern: in the e-commerce system, you can configure the delivery service to work in this mode. For redundancy and fault tolerance, you run multiple instances of the delivery service. It’s important that the physical goods are delivered only once, so only one instance of the delivery service should act on any given checkout message. The message interaction is asynchronous in this configuration, because shopping-cart doesn’t expect a response (ensuring delivery isn’t its responsibility!). You could use a message bus that provides work queues to implement this mode of interaction.

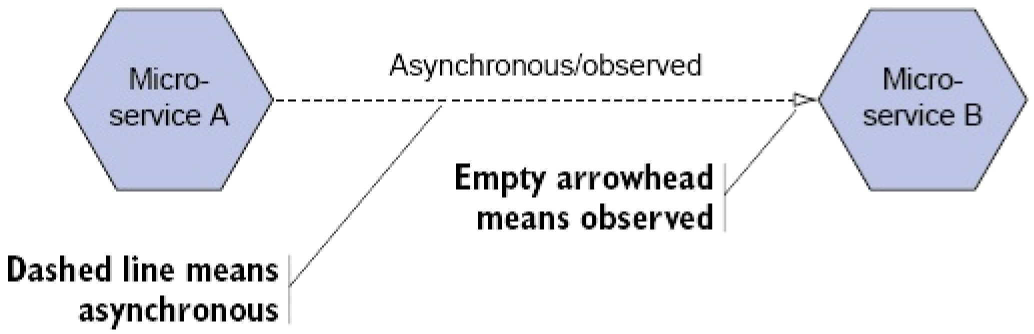


In reality, a message queue is a good mechanism for implementing this behavior, but it isn't absolutely necessary—perhaps you're using a sharding approach and ignoring messages that don't belong to your shard. As an architect using this pattern, you're again providing intent, rather than focusing on the network configuration of the microservices.

1/2: Fire-and-forget

This is another classic distributed pattern: publish/subscribe (see figure 3.5). In this case, all the listening microservice instances observe the message and act on it in some way.

Figure 3.5. Fire-and-forget pattern: the general form of this interaction involves a set of different microservices observing an emitted message. The shopping-cart service emits a checkout message, and various other services react to it: delivery, checkout-email, and perhaps audit. This is a common pattern



From a idealogical perspective, this is the purest form of microservice interaction. Messages are sent out into the world, and whoever cares may act on them. All the other patterns can be interpreted as constraints on this model.

Strictly speaking, figure 3.5 shows a special case: multiple instances of the *same* microservice will all receive the message. Diagrams of real systems typically include two or more listening services. This special case is sometimes useful because although it performs the same work more than once, you can use it to deliver guaranteed service levels. The catch is that the task must be idempotent.[55]

For example, the e-commerce website will display photos of products. These come in a standard large format, and you need to generate thumbnail images for search result listings. Resizing the large image to a thumbnail always generates the same output, so the operation is idempotent. If you have a catalog of millions of products (remember, your company is quite successful at this stage), then some resizing operations will fail due to disk failures or other random issues. If 2% of resizings fail on average, then performing the resizing twice using different microservice instances means only 0.04% (2% x 2%) of resizings will fail in production. Adding more microservice instances gives you even greater fault tolerance. Of course, you pay the price in redundant work. We'll examine this trade-off in chapter 8.

3.5.2 Core patterns: two messages/two services

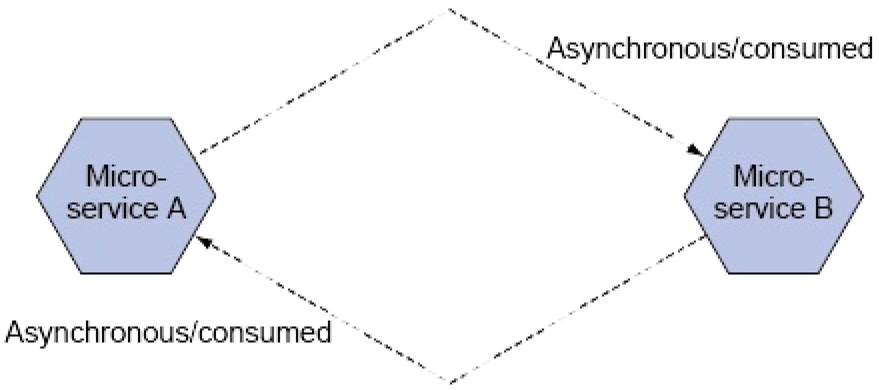
These message-interaction patterns capture causality between messages. These patterns describe how one type of message generates other types of messages. This is a deliberate change of perspective: traditionally, you'd think in terms of the dependency relationships between microservices, now you think about causality between messages.

From a messages-first perspective, it makes more sense. The causal relationships between messages are more stable than the relationships between the microservices that support the messages. It's easier to handle the messages with a different grouping of microservices than it is to change the message language.

2/2: Request/react

This is a classic enterprise pattern (see figure 3.6).[56] The requesting microservice enables the listening microservice to respond asynchronously by accepting a separate message in reaction to the initial request message. The requesting microservice is responsible for correlating the outbound request message and its separate response message. This is a more manual version of traditional request/response.

Figure 3.6. Request/react pattern

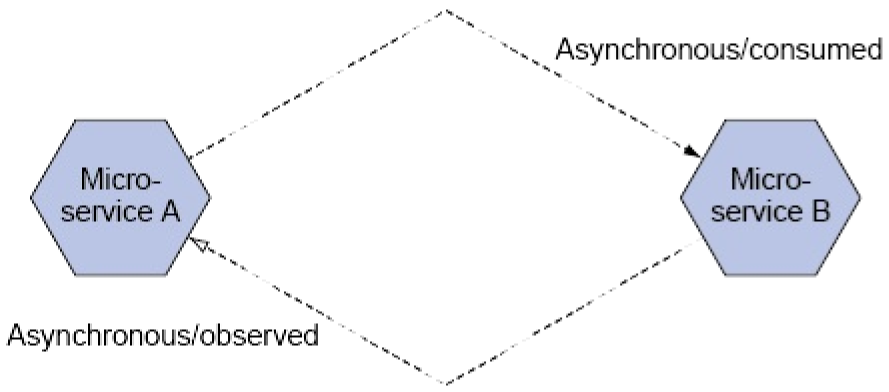


The advantage here is that you create temporal decoupling. Resources aren’t consumed on the requesting side while you wait for a response from the listener. This is most effective when you expect the listener to take a nontrivial amount of time to complete the request. For example, generating a secure password hash necessarily requires expending significant CPU time. Doing so on the main line of microservices that respond to user requests would negatively impact response times. Offloading the work to a separate set of worker microservices solves the problem, but you must allow for the fact that the work is still too slow for a normal request/response pattern. In this case, request/react is a much better fit.

2/2: Batch progress reporter

This a variant of the request/react pattern that gives you a way to bring batch processes into the fold of microservices (see figure 3.7). Batch processes, such as daily data uploads, consistency checks, and date-based business rules, are often written as programs that run separately from the main request-serving system. This arrangement is fragile, because those batch processes don’t fall under the same control and monitoring mechanisms.

Figure 3.7. Batch progress reporter



By turning batch processes into microservices, you can bring them under control in the same way. This is much more efficient from a systems management perspective. And it follows the underlying principle that you should think of your system in microservice terms: small services that respond to messages.

In this case, the message interaction is similar to request/react, but there's series of reaction messages announcing the state of the batch process. This allows the triggering microservice, and others, to monitor the state of the batch process. To enable this, reaction messages (in this case) are observed rather than consumed.

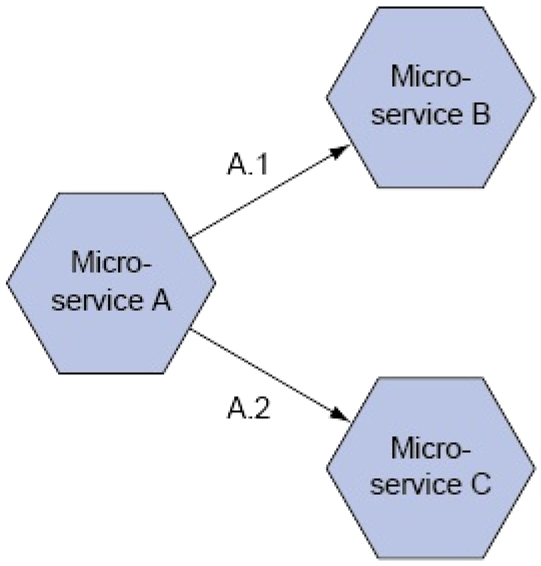
3.5.3 Core patterns: one message/n services

In these patterns, you begin to see some of the core benefits of using microservices: the ability to deploy code to production in a partial and staged way. In monolithic applications, you'd implement some of these patterns using feature flags or other custom code. That approach is difficult to measure and incurs technical debt. With microservices, you can achieve the same effect under the same simple model as all other deployments. This level of homogeneity makes it easier to manage and measure.

1/n: Orchestra

In this pattern, an orchestrating service coordinates the activities of a set of supporting services (see figure 3.8, which shows that microservice *A* interacts with *B* first, then *C*, in the context of some workflow *a*).

Figure 3.8. Orchestra

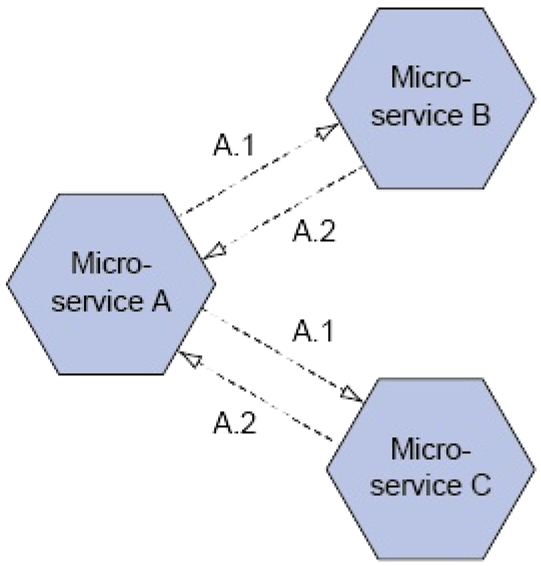


A criticism of the microservice architecture is that it’s difficult to understand microservice interactions, and thus service orchestration must be an important infrastructural support. This function can be performed directly by orchestration microservices that coordinate workflows directly, removing the need for a specialist network component to perform this role. In most large production microservice systems, you’ll find many microservices performing orchestration roles to varying degrees.

1/n: Scatter/gather

This is one of the more important microservice patterns (see figure 3.9).

Figure 3.9. Scatter/gather



Instead of a deterministic, serial procedure for generating and collecting results, you announce your need and collect results as they come in. Let's say you want to construct a product page for the e-commerce site. In the old world, you'd gather all the content from its providers and only return the page once everything was available. If one content provider failed, the whole page would fail, unless you'd written specific code to deal with this situation.

In the microservice world, individual microservices generate the content pieces and can't affect others if they fail. You must construct the result (the product page) asynchronously, because your content components arrive asynchronously, so you build in fault tolerance by default. The product page can still display if certain elements aren't ready—these can be injected as they become available. You get this flexibility as part of the basic architecture.

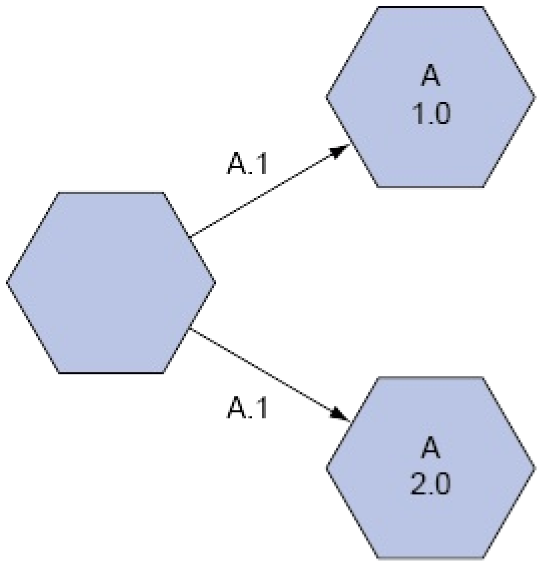
How do you coordinate responses and decide that your work is complete? There are a number of approaches. When you announce your need, you can set a time limit for responses; any response received after the time limit expires isn't included. This is the approach taken in the case study in chapter 9. You can also return once you get a minimum number of results.

For an extreme version, consider the sales tax use case. If you're willing to take availability over consistency, you can apply sales tax calculations and adapt to changes in rules in a flexible way. You announce that you need a sales tax calculation. *All* of the sales tax microservices respond, but you can rank the results in order of specificity. The more specific the result, the more likely it is to be the correct sales tax calculation, because it takes into account more information about the purchase. Perhaps this seems like a recipe for disaster and inaccurate billing. But consider that pricing and sales tax errors occur every minute of every day, and businesses deal with them by making compensating payments or absorbing the cost of errors. This is normal business practice. Why? Because companies prefer to stay open rather than close their doors—there's business to be done! We've allowed ourselves as software developers to believe that our systems must achieve perfect accuracy, but we should always ask whether the business wants to pay for it.

1/n: Multiversion deployment

This is the scenario we discussed in the section on the sales tax microservices earlier (see figure 3.10).

Figure 3.10. Multiversion deployment



You want to be able to deploy updated versions of a given microservice as new instances while keeping the old instances running. To do this, you can use the actor-style patterns (winner-take-all, fire-and-forget); but instead of distributing messages to a set of instances, all of which are the same version, you distribute to instances of differing versions. The figure shows some interaction, *a*, where messages are sent to versions 1.0 and 2.0 of microservice *A*.

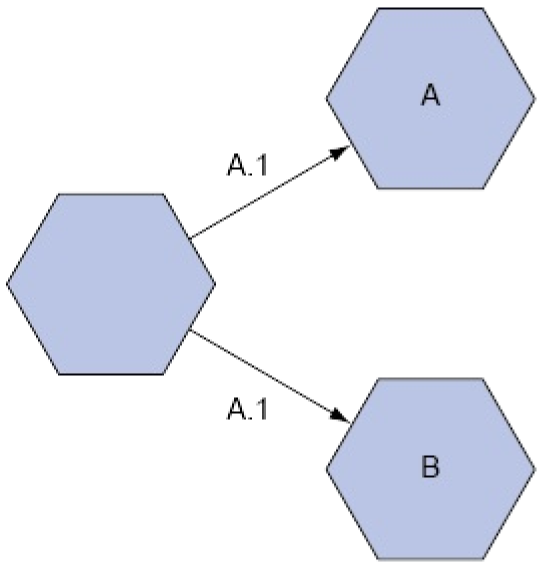
The facts that this deployment configuration is an extension of an existing pattern, and easy to achieve by changing the details of the deployment, shows again the power of microservices. We've converted a feature from custom implementation in code to explicit declarative configuration.

This pattern becomes even more powerful when you combine it with the deployment and measurement techniques we'll discuss in chapters 5 and 6. To lower the risk of deploying a new version of a service, you don't need to serve production results back from that new version. Instead, you can duplicate traffic so that the old versions of a microservice, proven in production, keep everything working correctly as before. But now you have the output from the new microservice, and you can compare that to the output from the old microservice and see whether it's correct. Any new bugs or errors can be detected using production traffic, but without causing issues. You can iterate deployment of the new version in this traffic-duplication mode until you have a high degree of confidence, using production traffic, that it won't break things. This is a fantastic way to enable high-velocity continuous delivery.

1/n: Multi-implementation deployment

Expanding the possibilities offered by multiversion deployment, you can do multi-implementation deployment (see figure 3.11).

Figure 3.11. Multi-implementation deployment



This means you can try out different approaches to solve the same problem. In particular, and powerfully, you can do A/B testing. And you can do it without additional infrastructure. A/B testing becomes a capability of your system, without being something you need to build in or integrate. The figure shows microservices *A* and *B* both performing the same role in message interaction *a*.

You can take this further than user interface A/B testing: you can A/B-test all aspects of your system, trialing new algorithms or performance enhancements, without taking massive deployment risks.

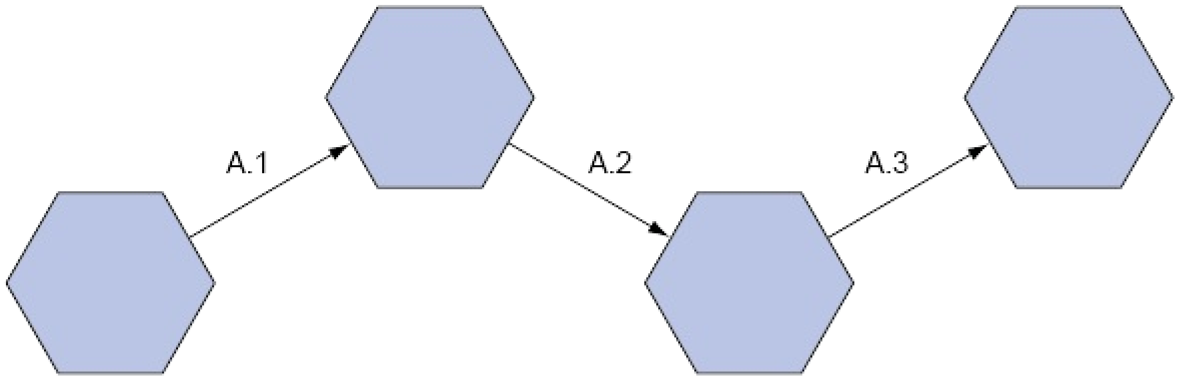
3.5.4 Core patterns: m messages/n services

The possibilities offered by configuration of multiple services and multiple messages expand exponentially. That said, bear in mind that all message patterns can be decomposed into the four core 1/2 patterns. This is often a good way to understand a large microservice system. There are a few larger-scale patterns that are worth knowing, which we'll explore here.

3.5.5 m/n: chain

The chain represents a serial workflow. Although this is often implemented using an orchestrating microservice (discussed in section 3.5.3), it can also be implemented in the configuration shown in figure 3.12, where the messages of the *a* workflow are choreographed;[57] the serial nature of the workflow is an emergent property of the individual microservices following their local rules.

Figure 3.12. Chain

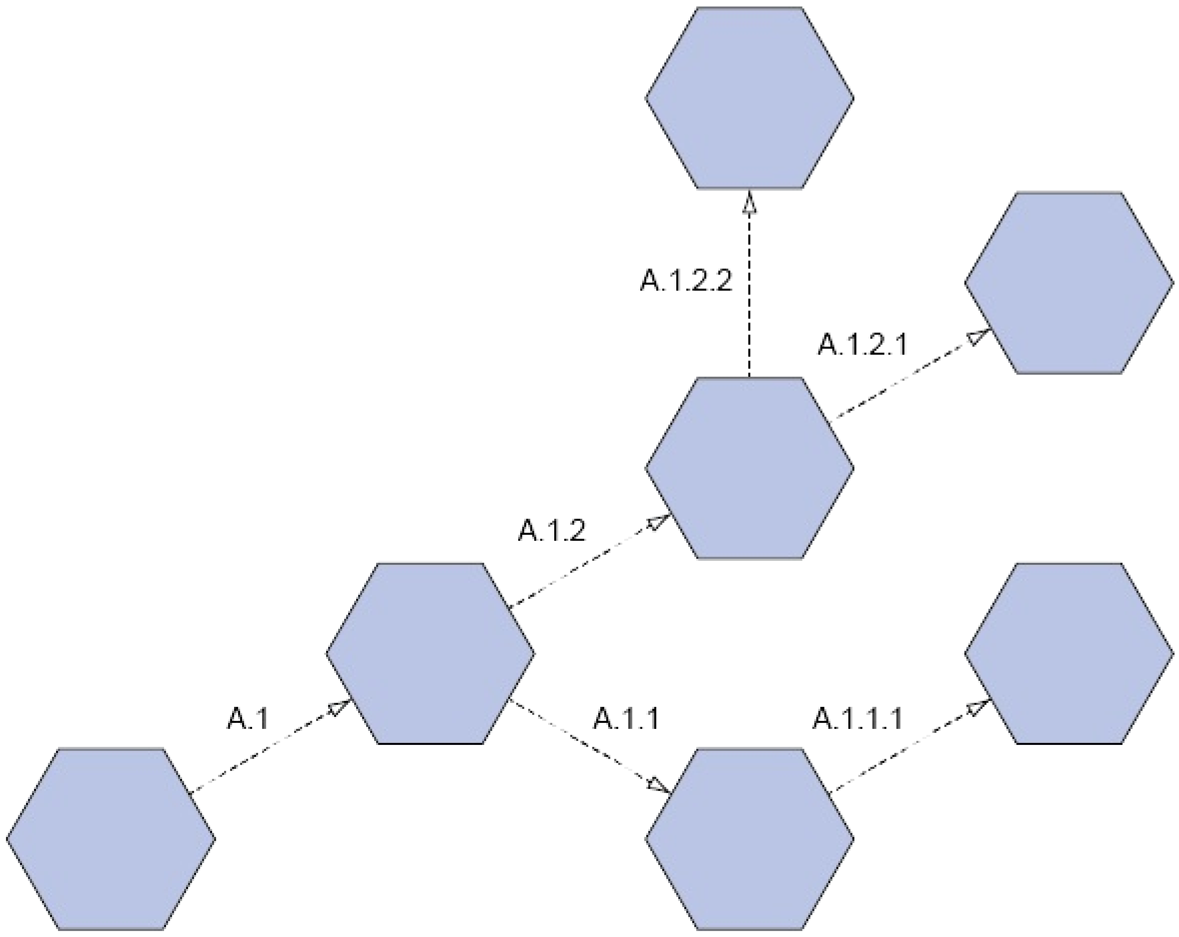


Generally, distributed systems that can perform work in parallel are always bounded by the work that can’t be parallelized. There’s always some work that must be done serially. In an enterprise software context, this is often the case where actions are gated: certain conditions must be met before work can proceed.

3.5.6 m/n: Tree

The tree represents a complex workflow with multiple parallel chains (see figure 3.13, which shows the message flow subsequences). It arises in contexts where triggering actions cause multiple independent workflows. For example, the checkout process of an e-commerce site requires multiple streams of work, from customer communication to fulfillment.

Figure 3.13. Tree



3.5.7 Scaling messages

There can be no question that the microservice architecture introduces additional load on the network and thus reduces performance overall. The foremost way to address this issue is to ask whether you have an actual problem. In many scenarios, the performance cost of microservices is more than offset by their wide-ranging benefits. You’ll gain a deeper understanding of how this trade-off works, and how it can be adjusted, in part 2 of this book.

To frame this discussion, it’s important to be specific about terminology:

- *Latency*—The amount of time it takes for the system to complete an action. You could measure latency by measuring the average response time of inbound requests, but this isn’t a good measure because it doesn’t capture highly variant response times. It’s better to measure latency using percentiles: a latency of 100 ms at the 90th percentile

means 90% of requests responded within 100 ms. This approach captures behavior that spikes unacceptably. Low latency is the desired outcome.
- *Throughput*—The amount of load the system can sustain, given as a rate: the number of requests per second. By itself, the throughput rate isn't very useful. It's better to quote throughput and latency together: such a rate at such a percentile.

High throughput with low latency is the place you want to be, but it isn't something you can achieve often. Like so much else in systems engineering, you must sacrifice one for the other or spend exponentially large amounts of money to compensate.

Choosing the microservice architecture means making an explicit trade-off: higher throughput, but also higher latency. You get higher throughput because it's much easier to scale horizontally—just add more service instances. Because different microservices handle different levels of load, you can do this in a precise way, scaling up only those microservices that need to scale and allocating resources far more efficiently. But you also get higher latency, because you have more network traffic and more network hops. Messages need to travel between services, and this takes time.

Lowering latency

When you build a microservices system with a messages-first approach, you find that certain message interactions require lower latency than others: those that must respond to user input, those that must provide data in a timely fashion, and those that use resources under contention. These message interactions, if implemented asynchronously, will have higher latency. To reduce latency in these cases, you'll have to make another trade-off: increase the complexity of the system. You can do this by introducing synchronous messages, particularly of the request/response variety. You can also combine microservices into single processes that are larger than average, but that increase in size brings with it the associated downsides of monolithic architectures.

These are legitimate trade-offs. As with all performance optimizations, addressing them ahead of time, before you have a proper measurement, is usually wasted effort. The microservice architecture makes your life easier by giving you the measurement aspect for a lower cost, because you monitor message flow rates in any case. Identifying performance bottlenecks is much easier.

It’s worth noting that lower-level optimizations at the code level are almost useless in a microservices context. The latency introduced by network hops swamps everything else by several orders of magnitude.

Increasing throughput

Achieving higher message throughput is easier, because it requires fewer compromises and can be achieved by spending money. You can increase the number of service instances, or use message queues[58] with better performance characteristics, or run on bigger CPUs.

In addition to adding muscle at the system level, you can make your life easier at the architectural level. Messages should be small: resist the urge to use them for transferring large volumes of data or large amounts of binary data, such as images. Instead, messages should contain references to the original data. Microservices can then retrieve the original data directly from the data-storage layer of your system in the most efficient manner.

Be careful to separate CPU-bound activity into separate processes, so it doesn’t impact throughout. This is a significant danger in event-based platforms such as a Node.js, where CPU activity blocks input/output activity. For thread-based platforms, it’s less of an issue, but resources are still consumed and have limits. In the sales tax example, as the complexity of the rules grows, computation time increases; it’s better to perform the calculation in a separate microservice. For highly CPU-intensive activities such as image resizing and password hashing, this is even more important.

[52] The open source toolset from Netflix is well worth exploring: https://netflix.github.io.

[53] The load balancer isn’t necessarily a standalone server. Client-side load

balancing has the advantage that it can be achieved with lightweight libraries and removes a server from the deployment configuration of your system.

[54] The power of the microservice idea is shown by the fact that in its simplest form it reduces to a monolith. We all know what monoliths reduce to.

[55] An idempotent task can be performed over and over again and always has the same output: for example, setting a data field to a specific value. No matter how many times the data record is updated, the data field always gets the same value, so the result is always the same.

[56] For more, see the excellent book SOA Patterns by Arnon Rotem-Gal-Oz, (Manning, 2012), https://www.manning.com/books/soa-patterns.

[57] Sam Newman, author of Building Microservices (O'Reilly, 2011), introduced the terms orchestration and choreography to describe microservice configurations.

[58] Kafka is fast, if that's what you need: http://kafka.apache.org.

3.6 When messages go bad

Microservice systems are complex. They don't even start out simple, because they require deployment automation from the beginning. There are lots of moving parts, in terms of both the types of messages and microservices and on-the-ground instances of them. Such systems demonstrate emergent behavior—their internal complexity is difficult to comprehend as a whole. Not only that, but with so many interacting parts, subtle feedback loops can develop, leading to behavior that's almost impossible to explain.

Such systems are discussed, in the general case, by Nassim Nicholas Taleb, author of the books *Black Swan* (Penguin, 2008) and *Antifragile* (Random House, 2014). His conceptual model is a useful framework for understanding microservice architectures. He classifies systems as fragile, robust, and antifragile:

- *Fragile* systems degrade when exposed to disorder and pressure. Most software falls into this category, failing to cope with the disorder generated by high loads.
- *Robust* systems can resist disorder, up to a point. They're designed to impose rigid discipline. The problem is that they suffer catastrophic failure modes—they work until they don't. Most large-scale software is of this variety: a significant investment is made in debugging, testing, and process control, and the system can withstand pressure until it hits a boundary condition, such as a schema change.
- *Antifragile* systems benefit from disorder and become stronger. The human immune system is an example. The microservice architecture is weakly antifragile by design and can be made more so by accepting the realities of communication over the network.

The key to antifragility is to accept lots of small failures, in order to avoid large failures. You want failures to be high in frequency but of low consequence. Traditional software quality and deployment practices, geared for the monolith, are biased toward low-frequency, high-consequence failures. It's much better for individual microservices to fail by design (rebooting themselves as needed) than for the entire system to fail.

The best strategy for microservices, when handling internal errors, is to fail fast and reset. Microservices are crash-first software. Instances are small and plentiful, so there's always somebody to take up the slack.

Dealing with external failure is more challenging. From the perspective of the individual microservice, external failures appear as misbehaving messages. When messages go bad, it's easy for them to consume resources and cause the system to degrade. It's therefore vital that the message-transportation infrastructure and the microservices adopt a stance that expects failure from external actors. No microservice should expect good behavior from the rest of the network. Luckily, many of the failure modes can be dealt with using a common set of strategies.

3.6.1 The common failure scenarios, and what to do about them

In this section, the failure modes are organized under the message interactions that are most susceptible to them. All failure modes can occur with any message interaction, but it’s useful to organize your thinking. All message interactions ultimately break down into the four core 1/2 interactions, so I’ll use the core interactions to catalog the failure modes.

For each failure mode, I’ll present options for mitigation. No failure mode can be eliminated entirely, and the principle of antifragility teaches that this is a bad idea anyway. Instead, you should adopt mitigation strategies that keep the whole healthy, even if a part must be sacrificed.

Note

In the following scenarios, microservice *A* is the sending service, and microservice *B* is the listening service.

3.6.2 Failures dominating the request/response interaction

The request/response interaction is vulnerable to high latency, which can cause excessive resource consumption as you wait for responses.

Slow downstream

Microservice *A* is generating work for microservice *B*. *A* might be an orchestrater, for example. *B* performs resource-intensive work. If *B* becomes slow for some reason, perhaps because one of its dependencies has become slow, then resources in *A* will be consumed waiting for responses from *B*. In an event-based platform, this will consume memory; but in a thread-based platform, it will consume threads. This is much worse, because eventually all threads will block waiting for *B*, meaning no new work can proceed.

Mitigation: use a circuit breaker. When *B* slows below a triggering level of throughput, consider *B* to be dead, and remove it from interactions with *A*. *B* either is irretrievably corrupt and will die anyway in due course and be replaced, or is overloaded and will recover once load is reduced. In either case, a healthy *B* will eventually result. The circuit breaker logic can operate either within *A* or in an intelligent load balancer in front of *B*. The net result is that throughput and latency remain healthy, if less performant than usual.

Upstream overload

Similar to the previous scenario, microservice *A* is generating work for microservice *B*. But in this scenario, *A* isn't well behaved and doesn't use a circuit breaker. *B* must fend for itself. As the load increases, *B* is placed under increasing strain, and performance suffers. Even if you assume that the system has implemented some form of automated scaling, load can increase faster than new instances of B can be deployed. And there are always cost limits to adding new instances of *B*. Simply blindly increasing the system size creates significant downside exposure to denial-of-service attacks.

Mitigation: *B* must selectively drop messages from *A* that push *B* beyond its performance limits. This is known as *load shedding*. Although it may seem aggressive, it's more important to keep the system up for as many users as possible. Under high-load scenarios, performance tends to degrade across the board for all users. With load shedding, some users will receive no service, but at least those that do will receive normal service levels.

If you allow your system to have supervisory microservices that control scaling in response to load, you can also use signaling from *B* to mitigate this failure mode. *B* should announce via asynchronous messages to the system in general that it's under too much load. The system can then reduce inbound requests further upstream from *A*. This is known as *applying backpressure*.

3.6.3 Failures dominating the sidewinder interaction

Failure in this interaction is insidious—you may not notice it for many days, leading to data corruption.

Lost actions

In this scenario, microservices *A* and *B* have the primary interaction, but *C* is also observing, performing its own actions. Any of these microservices can be upgraded independently of the others. If this introduces changes to message content or different message behavior, then either *B* or *C* may begin to fail. Because one of the key benefits of microservices is the ability to perform independent updates, and because this enables continuous delivery, this failure mode is almost to be expected. It’s often used as a criticism of the microservice architecture, because the naïve mitigation is to attempt coordinated deployments. This is a losing strategy that’s difficult to do reliably in practice.

Mitigation: measure the system. When *A* sends the message, it should be received by *B* and *C*. Thus, the outbound and inbound flow rates for this message must be in the ratio 1:2. Allowing for transient variations, this ratio can be used as a health indicator. In production systems, there are many instances of *A*, *B*, and *C*. Updates don’t replace all of *A*, for example; rather, following microservice best practice, *A* is updated in stages, one instance at a time. This allows you to monitor the message-flow-rate ratio to see whether it maintains its expected value. If not, you can roll back and review. This is discussed more fully in chapter 6.

3.6.4 Failures dominating the winner-take-all interaction

These failures can bring your system down and keep it down. Simple restarts won’t fix the problem, because the issue is in the messages themselves.

Poison messages

This is a common failure mode for actor-style distributed systems. Microservice *A* generates what it thinks is a perfectly acceptable message. But a bug in microservice *B* means B always crashes on the message. The message goes back onto the queue, and the next instance of *B* attempts to process it and fails. Eventually, all instances of *B* are stuck in a crash-reboot cycle, and no new messages are processed.

Mitigation: microservice *B* needs to keep track of recently received messages and drop duplicates on the floor. This requires messages to have some sort of identifier or signature. To aid debugging, the duplicate should be consumed but not acted on. Instead, it should be sent to a dead-letter queue.[59] This mitigation should happen at the transport layer, because it’s common to most message interactions.

Guaranteed delivery

Asynchronous messages are best delivered using a message queue. Some message-queue solutions claim to guarantee that messages will be delivered to listeners at most once, exactly once, or at least once. None of these guarantees can be given in practice, because it’s fundamentally impossible to make them. Message delivery, in general, suffers from the Byzantine Generals problem: it’s impossible to know for sure whether your message has been delivered.

Mitigation: skew delivery to prefer at-least-once behavior. Then, you have to deal with the problem of duplicate messages. Making behavior idempotent as much as possible reduces the effect of duplicates, because they then have no effect. As explained earlier in the chapter, idempotency refers to the property of a system where it can safely perform the same action multiple times and end up in the same state: for example, the sales tax calculation is naturally idempotent because it always returns the same result for the same inputs.

3.6.5 Failures dominating the fire-and-forget interaction

These failure modes are a reminder that the gods will always find our mortal plans amusing. They gave us brains just powerful enough to build machines that we can never fully comprehend.

Emergent behavior

As your system grows, with many microservices interacting, emergent behavior will occur. This may take the form of messages appearing that shouldn’t appear, microservices taking unexpected actions, or unexplained surges in message volume.

Mitigation: a microservice system isn’t a neural network, however appealing the analogy. It’s still a system designed to operate in a specific fashion, with defined interactions. Emergent behavior is difficult to diagnose and resolve, because it arises from the interplay of microservice behavior rather than from one microservice misbehaving. Thus, you must debug each case individually. To make this possible, use correlation identifiers. Each message should contain metadata to identify the message, because it moves between microservices; this allows you to trace the flow of messages. You should also include the identifiers of originating messages, because this will let you trace causality—see which messages generated further messages.

Catastrophic collapse

Sometimes, emergent behavior suffers from a feedback loop. In this case, the system enters a death spiral caused by ever-increasing numbers of unwanted messages triggering even more messages. Rolling back recent microservice deployments—the standard safety valve—doesn't work in this scenario, because the system has entered a chaotic state. The triggering microservice may not even be a participant in the problematic behavior.

Mitigation: the last resort is to bring the system down completely and boot it up again, one set of services at a time. This is a disaster scenario, but it can be mitigated. It may be sufficient to shut down some parts of the system and leave others running. This may be enough to cut the feedback loop. All of your microservices should include kill switches that allow you to shut down arbitrary subsets of the system. They can be used in an emergency to progressively bring down functionality until nominal behavior is restored.

[59] The dead-letter queue is the place you send copies of broken messages for later analysis. It can be as simple as a microservice that takes no actions but logs every message sent to it.

3.7 Summary

- One of the most important strategies for avoiding the curse of the distributed monolith is to put messages at the core your microservices thinking. With a messages-first approach, the microservice architecture becomes much easier to specify, design, run, and reason about.
- It's better to think about your business requirements in terms of the messages that represent them. This is an action-oriented stance rather than a data-oriented one. It's powerful because it gives you the freedom to define a language of messages without predetermining the

microservices that will handle them.

- The synchronous/asynchronous dichotomy is fundamental to the way messages interact. It’s important to understand the constraints that each of these message-interaction models imposes, as well as the possibilities the models offer.
- Pattern matching is the principle mechanism for deciding which microservice will act on which message. Using pattern matching, instead of service discovery and addressing, gives you a flexible and understandable model for defining message behavior.
- Transport independence is the principle mechanism for keeping services fully decoupled from the concrete topology of the network. Microservices can be written in complete isolation, seeing the world only in terms of inbound and outbound messages. Message transport and routing become an implementation and configuration concern.
- Message interactions can be understood along two axes: synchronous/asynchronous and observed/consumed. This model generates four core message-interaction patterns, which can be used to define interactions between many messages and microservices.
- The failure modes of message interactions can also be cataloged and understood in the context of this model.

4 Services

This chapter covers

- Refining the concept of microservices
- Exploring principal variants of the microservice architecture
- Comparing monoliths versus microservices
- Thinking of microservices as software components
- What functionality should be external to microservices

To understand the implications and trade-offs of moving to a new architecture, you need to understand how it differs from the old way of doing things, and how the new way will solve old problems. What are the essential differences between monolithic and microservice architectures? What are the new ways of thinking? And how do microservices solve the problems of enterprise software development?

A *microservice* is a unit of software development. The microservices architecture provides a mental model that simplifies the world at a useful level. The proposition of this book is that microservices are the closest thing yet to ideal software components. They're perfectly sized artifacts for fine-grained deployment into production. They're easily measured to ensure correct operation. The microservice attitude is the belief that this architecture delivers a fast, practical, and efficient way to create business value with software.

4.1 Defining microservices

The term *microservice* is inherently fuzzy, as a social effect of the increasing popularity of the architecture. When we, as software builders, use the term, we should be specific in our meaning. Much of the writing on microservices shares the same attitude toward software development but uses differing definitions of this key term. Weak definitions limit our thinking and provide an easy target for criticism from vested interests. Let's examine a sample of the proposed definitions:

- *Microservices are self-contained software components that consist of no more than 100 lines of code.* This definition captures the desire to keep microservices small and maintainable by one developer, rather than a team. It's an appeal to the idea that extreme simplicity has extreme benefits: 100 lines of code can be quickly and confidently reviewed for errors. [60] The small body of code is also inherently disposable in that it can easily be rewritten if necessary. These are desirable qualities for microservices, but not exhaustive. For example, the questions of deployment and interservice communication aren't addressed. The fundamental weakness in this definition is the use of an arbitrary numerical constraint that falls apart if we change programming languages. As we consider other definitions, let's retain the desire for code small enough to verify easily and to throw away if need be.
- *Microservices are independently deployable processes communicating asynchronously using lightweight mechanisms focused on specific business capabilities running in an automated but platform- and language-independent environment*, or words to that effect. On the opposite end of the spectrum are catchall general definitions. These definitions contain a laundry list of desired attributes. Are the attributes ordered by importance? Are they exhaustive? Are they well defined? General definitions give you a feeling that you're in the right galaxy, but they don't provide directions to design a microservice system. They invite endless semantic debate over the definitions of the attributes. What, for example, is a *truly* lightweight communication mechanism? [61] What we can take from these definitions is a working set of ideas that can be used in practice but that don't by themselves provide much clarity.
- *Microservices are mini web servers offering a small REST-based HTTP API that accepts and returns JSON documents.* This is certainly a common implementation. And these are microservices. But how big are they? And how does this definition address all the other concerns, such as independent deployability? Here we are too prescriptive on some questions and not prescriptive enough on others. Few would disagree that mini web servers are microservices. [62] And yet it excludes most of the interesting microservice architectural patterns, particularly those that

take advantage of asynchronous messages. This definition is not only weak but dangerous. Empirical evidence from the field suggests it often leads to tightly coupled services that need to be deployed together. [63] The takeaway from this failed definition is that limiting ourselves to thinking only in terms of web-service APIs prevents us from appreciating the radical possibilities that a wider concept can bring. A definition should provide power to our thinking, not constrain it.
- *A microservice is an independent software component that takes no more than one iteration to build and deploy*. In this definition, the focus is on the human side of the architecture. The phrase *independent software component* is suggestive and wide ranging, so this definition also attempts to be inclusive of implementation strategies. Microservices are software components, using the common understanding of the term. [64] This definition expresses the desire for microservices to indeed be "micro" by limiting the resources available to write them: one iteration is all you get. It also gives a nod to continuous delivery—you have to be able to deploy within an iteration. The definition is careful to avoid mention of operating system processes, networking, distributed computing, and message protocols; none of these are essential properties.[65]

We'll stop the definition game there. We must accept that we aren't school children, but professional software developers, and we live in the messy world of grownups. There's no tidy definition of microservices, and any definition we choose restricts our thinking. Rather than seek a definition that's dependent on numerical parameters, or that attempts to be exhaustive, or is too narrow, we should aim to develop a conceptual framework that is *generative*: the concepts within the framework generate an accurate understanding of the inherent trade-offs of the microservices architecture. We then apply these concepts to the context at hand to deliver working software. [66]

[60] C. A. R. Hoare, the inventor of the quicksort algorithm, in his 1980 Turing Award Lecture, famously said, "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

[61] It’s impossible to win a war of definitions. As soon as you provide a conclusive counter-example, your opponent denies that the counter-example is actually an example of the subject under discussion. The British philosopher Antony Flew provides the canonical example of this tactic, which can be paraphrased as follows: Robert: “All Scotsmen wear kilts!”; Hamish: “My uncle Duncan wears trousers.”; Robert: “Yes, but no true Scotsman does.”

[62] Cloud functions, in implementations such as AWS Lambdas and their kin, are often presented as the ideal way to build microservices, as they fit this particular definition very neatly.

[63] In my previous life as a consultant, I directed my poor teams to build many large systems this way, and we tied ourselves in the most wonderful Gordian knots.

[64] Software *components* are self-contained, extensible, well-defined, reusable building blocks. You are welcome to play definition games with the term "software component", but since we are here to discuss microservices, we shall have to take this one for granted.

[65] Erlang processes are most certainly microservices or, perhaps more correctly, nanoservices! You’re strongly advised to read Joe Armstrong’s Ph.D. thesis for the full details: “Making Reliable Distributed Systems in the Presence of Software Errors,” Royal Institute of Technology, 2003, erlang.org/download/armstrong_thesis_2003.pdf.

[66] Microservices are a subject worthy of an entire book, not a trite summary definition. But then, I would say that.

4.2 Microservice architectures

If we accept that microservices should communicate with each other using messages, and we want them to be independent, that implies that microservices must have a well-defined communication interface. Discrete messages are the most natural mechanism for defining this interface.[67]

Understanding that interservice communication can be specified in terms of messages leads to a more powerful way to understand the dynamic nature of microservices. At one level, you need to understand which services talk to which other services. In practice, this understanding is less useful than you may think. As the number of services grows, the number of connections does too, and it becomes difficult to visualize the full set of interactions. One way to mitigate this complexity is to take a message-focused approach to describing the system. Consider that services and messages are two aspects of the same structure. As we dicussed in the previous chapter, it’s often more useful to think about a microservice system in terms of the messages that pass through the system, rather than the services that respond to them. Taking this perspective, you can analyze the patterns of message interactions, find common patterns, and generate microservice architecture designs.

4.2.1 The mini web servers architecture

In the mini web servers architecture, microservices are nothing more than web servers that offer small REST interfaces. Messages are HTTP requests and responses. Message content is JSON or XML documents, or simple queries. This is an example of a synchronous architecture. HTTP requests require a response. We’ll take this as a starting point and then consider how to make these mini web servers more like software components.

Each microservice needs to know the location of other services that it wants to call. This is an important characteristic, and weakness, of mini web servers. When your system has just a few services, you can configure each service with the network locations of the other services, but this quickly becomes unmanageable as the number of services grows. The standard solution is a service-discovery mechanism.

To provide service discovery, you need to run a service in your system that keeps a list of all microservices and their locations on the network. Each microservice must query the discovery service to find the services it wants to talk to. Sadly, this solution has lots of hidden complexity. First, keeping the discovery service consistent with the real state of the world is non-trivial—writing a good discovery implementation is difficult. [68] Second, microservices need to maintain the knowledge of other services that they’ve obtained from the discovery service, and deal with staleness and correctness issues in this knowledge. Third, discovery invites tight coupling between services. Why? Consider that inside monolithic code, you need a reference to an object to call a method. Now you’re just doing it over the network—you need a network location and a URL endpoint. If you do use service discovery, you introduce the need to provide infrastructure code and modules for your services to interact with the discovery mechanism.

In its simplest configuration, this architecture is point-to-point. Microservices communicate directly with each other. You can extend this architecture with more-flexible message patterns by using intelligent load balancing. To scale a given microservice, place an HTTP load balancer[69] in front of a set of microservice instances. You’ll need to do this for each microservice you want to scale. This increases your deployment complexity, because you’ll need to manage the load balancer configurations as well as your microservices.

If you make your load balancer intelligent, you can start to get some of the deeper benefits of microservices. Nothing says all the microservices behind a given load balancer need to be the same version of the same microservice. You can partially deploy and test new versions of a microservice in production by introducing it into the balance set. This is an easy way to run multiple versions of the same microservice at the same time.

You can place different microservices behind the same load balancer and then use the load balancer to pattern-match on the properties of inbound messages to assign them to the correct type of microservice.[70] Consider the power this gives you—you can extend the functionality of your system by adding a new microservice and updating the load balancer rules. No need to change, update, redeploy, or otherwise touch other running services. The ability to make these kinds of small, low-impact, low-risk production changes is a large part of the attraction of the microservice architecture. It makes continuous delivery of code to production much more feasible.

The load balancer doesn’t have to be a separate process in front of the listening microservices. You can use a client-side library, embedded in the client microservice, to perform the intelligent load balancing. The advantage is that you don’t have to worry about deploying and configuring lots of load balancers in your network. And the client-side load balancer can use service discovery to determine where to send balanced messages.

Cloud Functions

All the major cloud computing vendors provide a solution for deploying mini web servers, in the form of small synchronous functions in your mainstream programming language of choice. These vendors also provide "intelligent" load balancing and mechanisms for asynchronous message passing. Since the first edition of this book, cloud functions have become the safest and most popular method of adopting the microservice architecture. Given the wealth of documentation and third-party tooling now available, you will find implementing the patterns described in this book using cloud functions to be relatively problem-free. Sadly, they are not suitable for all scenarios and business contexts. This book very intentionally does not presume any given deployment implementation, so that you can adapt the patterns described here to your own situation.

4.2.2 The distributed objects architecture

This is the more classical approach to microservices, used before the term was coined or the hype started. If you’ve ever worked with CORBA, Enterprise Java Beans, or any message queues, your pain has been part of the inspiration for microservices.

This architecture emphasizes asynchronous messages, and message routing configurations. Your microservices subscribe to message queues and react to inbound messages by generating new outbound messages, rather than generating reponses. At the edges you still need to support synchronous interactions (with a web-based user interface, or an external web services API).

Your microservices will tend to be larger, and you will deploy as virtual machines, or containers. You’ll spend quite a bit of time doing network configuration, and if you’ve been very bad in a previous life, you’ll also have to configure an Enterprise Service Bus.

Local development will be tricky. You’ll either have to mock many of the services, or try to run them all locally using containers, or assign cloud servers (either shared, or if you’re lucky, and have money to burn, individually).

These practical difficulties were very common in the days before cloud functions became ubiquitous, and drive much of the negative discussion on microservices. The complaints are entirely justified and you end up in serious trouble if you don’t design your system, from production all the way down to the individual developer, very carefully.

The advantage of the distributed objects architecture is the level of control you gain. With the right tooling [71] you can solve most of the infrastructural pain. You get far more options for deployment (such as the ability to deploy to bare metal, or on-premise). A great many proto-microservice systems use this architecture, and a great many deliberate microservice systems also choose to do so, because it provides the flexibility needed. Is is also quite compatible with the mini web servers architecture, and many systems are hybrids of both.

4.2.3 The language platform architecture

Why cobble together a microservices system when you get one "batteries included". As I note many times in this book, microservices are just the latest attempt to solve the problems of software component design.

Another way to solve the problem is to bake microservices into your programming language platform. The language syntax and semantics, and the runtime implementation provide a structure that lets you write small, scalable, independently deployable and self-contained pieces of code—sound familiar?

Consider the following language platforms, and what their creators had to say about them:

- Erlang (OTP): Originally designed as a system for very high availability telecommunications systems, it is now used in many other contexts, and has more modern language variants such as Elixir. Start here: erlang.org

In Erlang, you have as many processes you want. You can arrange the processes observing each other … Processes are isolated by design. Context switching is very lightweight. The processes by design cannot damage each other.

-- Joe Armstrong *Erlang creator*

- Clojure: A modern LISP for the Java Virtual Machine, which also comes in a scripting variant. While not explicitly designed for microservices, many microservice practioners from the Java ecosystem find Clojure and its infrastructure to be a good fit for the microservices mental model.

I think, even if you set concurrency aside, using those kind of data structures and taking a functional approach to writing your programs, is gonna give you much, much better programs. Much more reliable, much easier to test and understand and maintain. BUT, when you put concurrency in the loop, there is no longer any contest: nothing compares to using this kind of a strategy in designing your program.

-- Rich Hickey *Clojure creator*

- SAP R/3 and it descendants: What from the outside looks like the epitome of monolithic enterprise software is actually a microservice architecture in all but name—Small, controlled units of functionality, designed to scale and be independently deployable. While you must deal with all the ills of a large legacy platform, it does demonstrate the power of focusing on software components as the defining structure in your system, even if that did result in the ABAP language!

The software industry has to become better in componentization. That’s a clear focus for most of the software companies. How components look, how they are maintained, the ability to maintain them separately.

-- Hasso Plattner *SAP Co-founder*

- COBOL and the Mainframe universe: these many legacy systems, primarily from IBM, remain in use and are a critical part of the financial, medical and defense infrastructure we rely on for our civilization. As with SAP, the underlying architecture has many microservice analogues.

In pioneer days they used oxen for heavy pulling, and when one ox couldn’t budge a log, they didn’t try to grow a larger ox. We shouldn’t be trying for bigger computers, but for more systems of computers.

-- Admiral Grace Hopper *COBOL creator*

- Smalltalk: this language and platform, and its theoretical basis, provide so much of the inspiration for the microservices architecture, that we must include it as a historical homage, even if would not be your first choice today (perhaps it is!). Although object-oriented programming in practice deviated far from Alan Kay’s original concept, the fundamental idea of objects communicating by messages as the basis for a component architecture is the starting point for using microservices.

Just a gentle reminder that I took some pains at the last OOPSLA to try to remind everyone that Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I’m sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging"—that is what the kernal [sic] of Smalltalk/Squeak is all about.

-- Alan Kay *Smalltalk creator*

The trade-off of these systems, which I am sure you have already noticed, is that you don’t get to choose the language, and you have to live with whatever constraints the platform imposes.

No-code and low-code Platforms

Recently we have seen the rise of platforms that promise the ability to build complex applictions with (almost) no-code. This is greatly to be applauded! As a coder you can move onto the more interesting work instead of building yet another CRUD application.

These platforms are the modern version of SQL, a language designed for non-technical users to empower report building. Fundamental to the implementation of these platforms is a component-based extension system of some kind. This is "microservices-in-a-box", which is something that will make all our lives easier.

There is a long way to go, and SQL never did quite make it out of developers hands, showing that coding is harder than it looks. But if these platforms can provide better developer experience (coding inside a HTML `textarea` is not fun), they may well come to dominate some areas of software development.

4.2.4 The messages before services architecture

The premise of this book is that you should focus on message first, using them to express your problem space. Only then do you move to services, allocating the messages as needed. As the system evolves, you move message handlers from one service to another, as a refactoring. You split messages, you combine messages, you make them synchronous, or asynchronous, as needed. You move from initial general cases to specific edge cases.

The construction of services, the routing and transport of messages, the data persistence layer, monitoring and scaling, are all implementation details in this architecture. You should be able to run your microservices as a local monolith (they’re just software components, so load them as modules!), a distributed system of containers (in Kubernetes, say), or a deployment of serverless functions. All with same, unchanged, business logic code.

Since this section is the entire contents of this book, I will stop here and refer you to remaining chapters.

[67] This doesn’t exclude other communication mechanisms such as streaming data, but these are generally special cases or used as a transport layer for embedded messages.

[68] Some relatively robust service discovery implementations are available: ZooKeeper (zookeeper.apache.org), Consul (consul.io), etcd (github.com/coreos/etcd), and others. None of them deliver fully on the fault-tolerance and data-consistency claims they make, although all are suitable for production. Check out Kyle Kingsbury's "Jepsen" series of articles at aphyr.com/tags/jepsen for detailed analysis.

[69] Suitable load balancers are NGINX (nginx.org), HAProxy (www.haproxy.org), and Eureka (github.com/Netflix/eureka).

[70] One way to do this is to use extension modules for servers such as NGINX. It's also perfectly workable to roll your own, using a platform such as Node.js (nodejs.org).

[71] Such as all the wonderful toys a company like Netflix has open-sourced: github.com/netflix

4.3 Monolithic projects vs. microservice projects

The monolithic software architecture creates negative consequences, which many have assumed are fundamental challenges that apply to all software development. On closer examination, with critical questioning, this assumption can be turned on its head. Many of the challenges, and many of the supposed solutions to these challenges, arise directly from the engineering effects of monolithic architecture and become moot when you take a different engineering approach.

There are three consequences of the monolith. First, all members of the software development team must carefully coordinate their activities so as not to block each other. There's one code base. If a single developer breaks the build, then all developers are at risk of blockage. The code naturally tends toward deep, multidimensional dependency. This is a consequence of perceived best practices. Refactoring common code into shared libraries creates deep dependency trees. The amount of rework needed when changing the code structure is exponentially proportional to the depth of that structure in the dependency tree. Teams often make the rational choice to wrap complexity in ever more layers, in an attempt to hide and contain it. Monoliths make the cost of parallel work much higher and thus slow development.

The second consequence of monoliths is that they gather technical debt rapidly. There’s no natural limiting force. A well-structured, properly decoupled, clean, object-oriented initial design is too weak to resist the immediate needs of an entire team working against the clock to deliver today’s features. There are too many ways that one piece of code can invade other pieces of code—too many ways to create dependencies.

A primary example is data-structure corrosion. Given an initial requirements definition, the senior developers and architects design appropriate data structures to describe the problem domain. These are shared data structures and must accommodate all known requirements and anticipate future requirements, so they tend toward the complex. Deeply nested cross-referencing is a tell-tale sign of attempted future proofing. Unfortunately, the world often outwits our meager intelligence, and the data structures can’t accommodate real business needs as they emerge. The team is forced to introduce kludges, implicit conventions, and ad hoc extension points.[72] Later, new developers and junior developers, lacking understanding of the forces on the data structure, may introduce subtle and devious bugs, costing the team vastly disproportionate time in fixes and performance-tuning workarounds.[73] Eventually, the team must engage in extensive refactoring efforts to regain some measure of development velocity.

Finally, the third consequence of monoliths is that they are all-or-nothing deployments. You have an old version of a monolith running in production, and you have a new version on staging. To upgrade production without impacting the business, you have a very stressful weekend ahead of you.

Perhaps you’re more sophisticated and use blue-green deployments to mitigate risk.[74] You still have to expend energy building the blue-green infrastructure, and it’s still not much help if you have database schema migrations, because those aren’t easily reversible.

The basic problem is that any change to production requires a full redeployment of the entire code base. This creates high-risk exposure to failures at all levels of the system. No amount of unit testing, acceptance testing, integration testing, manual testing, and trialing can give you a true measure of the probability of failed deployment, because the failure conditions are often direct consequences of production conditions that you can't simulate. Production data (which you may not even have access to, due to client confidentiality rules) only needs one unforeseen aspect to cause critical failures. It's difficult to verify performance using test data—production can be orders of magnitude larger. Users can behave in unanticipated ways, especially with new features, that break the system because the team wasn't able to imagine those use cases. The deployment risk associated with monoliths causes slow, infrequent releases of new features, holding back fast delivery.

These engineering challenges, for that is precisely what they are, can't be solved with any given project management approach. They're in a different problem domain. And yet, almost universally and exclusively, businesses try to solve them with software development methodologies and project management techniques. Rather than stepping back and searching for the real reason projects are delivered late and over budget, software development stakeholders blame themselves for poor execution. No amount of good execution will let you build skyscrapers with bullshit.[75] The solution lies elsewhere.

The microservices architecture, as an engineering approach, allows us, as software developers, to revisit all of our cherished best practices and ask whether they really make delivery faster and more predictable. Or are they merely poor mitigations of the fundamental problems of monolithic development? Frederick P. Brooks, in his seminal 1975 book *The Mythical Man-Month*, explains in graphic detail the challenges of monolith development.[76] He then suggests a set of techniques and practices, not to solve the problem, but to contain and mitigate it. This is the core message of the phrase "no silver bullet": no project management techniques can overcome the engineering deficits of the monolithic architecture.

4.3.1 How microservices change project management

The engineering features of the microservice architecture have a direct impact on the amount of project management effort needed to ensure successful delivery. There’s less need for detailed task management and for much of the useless ceremony of explicit methodologies.[77] Project management of microservice projects can use a light touch. Let’s work through the implications.

4.3.2 Uniformity makes estimation easier

Microservices are small, and a good practice is to limit them to at most one iteration’s worth of work from one developer, for the initial version. Microservice estimation is thus a much easier task than general software-effort estimation, because you force yourself to chunk features into iteration-sized bites. This is an important observation. Traditional monolithic systems are composed of heterogeneous components of various sizes and complexity. Accurate estimation is extremely difficult, because each component is a special case and has a multifaceted set of interactions with other components via method calls, shared objects and data structures, and shared database schemas. The result is a project task list that bends to the demands of the system architecture.[78]

With microservices, the one-iteration complexity limit forces uniformity on components that increases estimation accuracy. In practice, even more accuracy can be achieved by classifying microservices into, say, three complexity levels, also classifying developers into three experience levels, and matching microservices to developers. For example, a level-1 microservice can be completed by a level-1 developer in one iteration, whereas a level-3 microservice needs a level-3 developer to be completed in one iteration. This approach gives far more accuracy than generic agile story point estimates that ignore variations in developer ability. A microservice project can be accurately planned using a sensible, meaningful mapping from microservices to iterations. We’ll explore this idea in more detail in part 2 of this book.

Why is software estimation difficult?

Why is it so difficult to estimate the complexity of components of a larger system? The tight coupling that invariably occurs in monolithic architectures means development in the later stages of a project is exponentially slower, making the initial estimates of late-stage components highly skewed toward the overoptimistic. The exponential slowness arises from the mathematical fact (known as Metcalfe’s law) that the number of possible connections between nodes in a network increases proportionally to the square of the number of nodes.

And there’s another factor: human psychology suffers from many cognitive biases—we’re not good at working with probabilities, for instance. Many of these come into play to sabotage accurate project estimation. Just one example: *anchoring* is the bias for staying close to the first number you hear. The complexity and thus completion time for software components follow a power law: most take a short amount of time, but some take much longer.[79]

The largest and most difficult components are underestimated, because the bulk of the estimation work concerns small components. The old joke that the last 10% of the schedule takes 90% of the time expresses much truth.

4.3.3 Disposable code makes for friendlier teams

Microservice code is disposable. It’s literally throw-away. Any given microservice is one iteration’s worth of work, for one developer. If the microservice was badly written, is underperformant in the chosen language, or isn’t needed anymore because requirements have changed, then it can be decommissioned without much soul searching. This reality has a healthy effect on team dynamics: nobody becomes emotionally attached to their code, nor do they feel possessive of it.

Suppose Alice thinks microservice A, written by Bob several iterations back in Java, will be twice as performant if written in C++. She should go for it! It’s an extra iteration invested either way, and if the attempt is a failure, the team is no worse off, because they still have Bob’s Java code.

The knowledge that each microservice must live or die on its own merits is a natural limiting function for complexity. Complexity makes you weak. Better to write a new, special-case microservice than extend an existing one. If you reserve, say, 20% of iterations for rewrites and unforeseen special cases, you can have more confidence that this is real contingency, rather than a political tactic in the effort-negotiation game.[80]

4.3.4 Homogeneous components allow for heterogeneous configuration

You can group microservices into different classes with differing business constraints. Some are mission critical and high load—for example, the checkout microservice on an e-commerce website. Others are core features, but not mission critical. Still others are nice-to-haves, where failure has no immediate impact. Questions: Is it necessary for all of these different kinds of microservices to have the same quality level? Do they all need the same level of unit-test coverage? Does it make sense to expend the same quality control effort uniformly on all microservices? No. It is a waste of scarce resources to enforce uniform quality.

You should expend effort where it counts. You should have different levels of unit-test coverage for different classes of microservice. Similarly, there are different performance, data-safety, security, and scaling requirements. Monolithic code bases have to meet the highest levels across the board, because it’s far too complicated to do otherwise. Microservices allow a finer-grained focus on applying limited developer resources where they count.[81]

Microservices make successful failures successful. The typical software system must often pass user-acceptance testing. In practice, this means the entity with the checkbook won’t sign until a set of features has been ticked off. Step back for a minute, and ask yourself whether this is a good way to ensure that the delivered software will meet the business goals for which it was originally commissioned. How can anyone be sure that a given feature delivers actual value until it’s measured in production? Perhaps certain features will never be used or are overly complex. Perhaps you’re missing critical features nobody thought of. And yet user-acceptance testing treats all features as having the same value. In practice, what happens is that the team delivers a mostly complete system with a mostly random subset of the originally desired features. After much grumbling, this is accepted, because the business needs the system to go into production.

A microservice approach doesn't change the reality that developer resources are limited and ultimately there may not be enough time to build everything. It does let you take a breadth-first approach. Most projects take a depth-first approach: user stories are assigned to iterations, and the team burns down the requirements. At the end of the original schedule, this leaves you with, say, 80% of the features completed and 20% untouched. In a breadth-first approach, you deliver incomplete versions of all features. At the end of the project, you have 100% of features mostly complete, but a substantial number of edge cases aren't finished. Which of these is the better position to be in for go-live? With the breadth-first approach, you cover all the cases the business people thought of, at some level. You haven't wasted effort fully completing features that will turn out to have no value. And you've given the business the opportunity during the project to redirect effort without giving up on entire features—a much easier discussion to have with stakeholders. Microservices make allocation of finite development resources more efficient and friendly.

4.3.5 There are different types of code

Microservices allow you to separate business-logic code from structural code. Business-logic code is driven directly from business requirements. It's determined by the best guesses of the business stakeholders, given incomplete and inadequate business information. It's naturally subject to rapid change, hidden depths, and obsolescence. Corralling this business-logic code into microservice units is a practical engineering approach to managing rapid change.

There's another type of code in the system: structural code. This is where system integration, algorithms, data-structure manipulation, parsing, and utility code happen. This code is less subject to the vagaries of the business world. There's often a relatively complete technical specification, an API to work against, or specifically limited requirements. This code can safely be kept separate from the business-logic code (in shared libraries, or core microservices), so it neither slows down business code nor is negatively impacted by incidental business logic.

The problem with most monolithic architectures is that these two types of code—business-logic and structural—end up mixed together, with predictably negative effects on team velocity and levels of technical debt. Business logic belongs in microservices; structuring belongs in software libraries. The ability to allocate coding effort correctly in this way makes estimating the level of effort required for each more accurate, and increases the predictability of the project schedule.

[72] Declarative structures are usually the best option for representing the world, because they can be manipulated in repeatable, consistent, deliberately limited ways. If you introduce ways to embed executable code to handle special cases as a “get out of jail free card,” things can get complicated fast, and you end up in technical debtors’ prison anyway.

[73] An example from a former client is instructive: The client added a database column for XML content so that they could store small amounts of unstructured data. The schema for that XML included several elements that could be repeated, to store lists. These lists were unbounded. In the problem domain, a small number of users generated very long lists, leading to massive XML content, leading to strange and wonderful garbage collection issues that were long separated from the root cause.

[74] The blue-green deployment strategy means you have two versions of the system in production at all times: the blue version and the green version. Only one of them is live. To deploy, you upgrade the offline version and swap.

[75] Wattle-and-daub construction has been used since Neolithic times and is an excellent construction technique that can get you to three or four small stories, given a sufficiently large herd of cattle to generate excrement. It won’t help you build the Empire State Building.

[76] Brooks was the manager for the IBM System/360 mainframe project and the first to make the written observation that adding more developers to an already-late project just makes it even later.

[77] To spare embarrassment, no methodologies will be named. But you know who you are. If there were a project management approach to software

development that could consistently deliver, over many kinds of teams, we’d already be narrowing down toward the solution. But we see no signs of significant progress.

[78] Somewhat ironically, Fibonacci estimation (where agile story point estimates must be Fibonacci numbers: 1, 2, 3, 5, 8, 13, …) is proof enough that a local maximum has been reached in the estimation accuracy of monolithic systems.

[79] Power laws describe many phenomena where small causes can have outsize effects: earthquake durations, executive salaries, and letter frequencies in text, for example.

[80] Software project estimation often deteriorates into a political game. Software developers give optimistic estimates to get gold stars. Business stakeholders, burned before by failed projects, forcefully demand all features on an arbitrary schedule. The final schedule is determined by horse-trading rather than engineering. Both sides have legitimate needs but end up in a lose-lose situation because it isn’t politically safe to communicate these needs.

[81] Chapter 9 discusses a way to quantify these fine-grained measurements.

4.4 The unit of software

The preceding discussion makes the case that microservices are incredibly useful as compositional units of software. Can they be considered fundamental units, much like objects, functions, or processes? Yes, because they give us a powerful conceptual model for thinking about system design.

The essence of the problem we’re trying to solve is one of multidimensional scaling: scaling software systems in production, scaling the complexity of the software that makes up those systems, and scaling the teams of developers that build them. The power of the microservices concept comes from the fact that it offers a unified solution to many different scaling problems.

Scaling problems are difficult because they’re exponential in nature. There are no 12-foot-tall humans, because doubling height means you increase body volume 8-fold, and the materials of our bodies, and our body architecture, can't handle the increased weight.[82] Scaling problems have this characteristic. Increasing one input parameter linearly causes disproportionate accelerated change in other aspects of the system.

If you double the size of a software team, you won't double the output speed. Beyond more than a few people, you'll move even slower as you add more people.[83] Double the complexity of a software system, and you won't double the number of bugs; you'll increase them by the square of the size of the code base. Double the number of clients you need to serve, and suddenly you need to manage a distributed system in order to handle the load.

Scaling can be addressed in two principal dimensions:[84] the vertical and the horizontal. *Vertical scaling* means making what you have bigger, stronger, or faster. This works until the physical, mathematical, or functional aspects of the system reach their fundamental limits. Thus, you can't keep buying more-powerful machines. *Vertical scaling* tends to have exponential decay in effectiveness and exponential growth in cost, which gives it hard limits in practice. That said, don't be afraid to scale vertically when you can afford it —hardware is much cheaper than developers.

Horizontal scaling escapes hard limits. Instead of making each piece more powerful, just keep adding more pieces. This has no fundamental limits, as long as your system is designed to be linearly scalable. Most aren't, because they have inherent communication limits that require too many pieces to talk to too many other pieces.

Biological systems comprising billions of individual cells have overcome horizontal-scaling limits by making communication as local as possible. Cells only communicate with their close neighbors, and they do so asynchronously using pattern matching on undirected hormonal signals. We should learn a lesson from this architecture!

High-capacity scaling arises when the system is composed of large numbers of independent homogeneous units. Sound familiar? The principal qualities of microservices lend themselves powerfully to effective scaling—not just in terms of load, but also in terms of complexity.

[82] You also need to double width and depth, to maintain proportions; hence, 2 cubed.

[83] Amazon has a scientific rule for the size of a software team: it must be possible to feed the entire team with no more than two pizzas.

[84] You can add dimensions and get scale cubes and scale hypercubes. This lets you refine your analysis, but two dimensions will do just fine for decision making.

4.5 Microservices are software components

In this book, I claim that microservices make excellent—almost perfect—software components. It’s worth examining this claim in more detail. Software components have relatively well-defined characteristics, and there’s broad general agreement on the most important of them. Let’s see how microservices stack up against this understanding.

4.5.1 Encapsulated

Software components are self contained. They encapsulate a set of semantically consistent activities and data. The outside world isn’t privy to this internal representation and can’t pollute it. Likewise, the component doesn’t expose its internal implementation. The purpose of this characteristic is to make components interchangeable.

Microservices deliver on encapsulation in a very strong way—far stronger than language constructs such as modules and classes. Because each microservice must assume a network separation from other microservices, it can only communicate with them via messages, and it has no backdoor access to the internals of other microservices. Creating such backdoor access requires more effort for developers, so encapsulation is strongly preserved throughout the lifetime of the system.

4.5.2 Reusable

Reusability is a holy grail of software development. A good component can be reused in many different systems over a long period of time. In practice, this is difficult to achieve, because each system has different needs. The component evolves over time and so has different versions. Reusability also implies extensibility: it should be easy to reuse the component in a new context without always needing to modify it. The purpose of this characteristic is to make components useful beyond a single project.

Microservices are inherently reusable, because they’re network services that can be called by anyone. There’s no need to worry about code integration or library linking. Microservices address the versioning and extensibility requirement not by enhancing the capabilities of the individual microservice,[85] but by allowing the system to add new special-case microservices and then using message routing to trigger the right service. We’ll talk about this in detail in chapter 7.

4.5.3 Well-defined interfaces

The interface offered by a component is the full definition of its contract with the outside world. This interface should have sufficient detail (but no more!) to allow the component to be interchangeable with other implementations and with other systems. The purpose of this characteristic is to enable free choice of components.

Microservices use messages, and only messages, to communicate with the outside world. These messages can be explicitly listed and their contents constrained as needed.[86] Microservices have well-defined (but not necessarily strict) interfaces by design.

4.5.4 Composable

The real power of components to accelerate software development comes not from reusability, which is merely a linear accelerator, but from combining components to do far more interesting things than each component can do separately. Components can be composed together into more capable components that themselves can be composed into larger systems.[87] The purpose of this characteristic is to make software development predictable by declaring the behavior of the system rather than constructing it.

Microservices are easily composable because the network flow of messages can be manipulated as desired. For example, one microservice can wrap another by intercepting all of the latter’s messages, modifying them in some way, and passing them on to the wrapped service.

[85] Traditionally, component systems rely on API hooks for extensibility. This is an inherently nonscalable approach because it isn’t homogeneous—every component and API hook is different.

[86] Resist the temptation to use message schemas and to enforce contracts between services. Doing so may seem like a good idea at the time, until you find yourself painted into a corner by your perfect schema. Microservices are for messy business logic, where strict schemas die every day.

[87] The most successful component architecture is UNIX pipes. By constraining the integration interface to be streams of bytes, individual command-line utilities can be composed into complex data-processing pipelines. The compositional power of this architecture is a major reason for the success of the operating system.

4.6 The internal structure of a microservice

The primary purpose of a microservice is to implement business logic. You should be able to concentrate on this purpose. You shouldn’t have to concern yourself with service discovery, logging, fault tolerance, and other standard behaviors; those are perfect candidates for framework or infrastructure code.

Microservices need a communications layer for messages. This should completely abstract the sending and receiving of messages, and the knowledge of where those messages need to go. As soon as one microservice knows about another, you have coupling, and you’re on a slippery slope to a fragile system. Message delivery should be transport independent: messages can travel over any medium, whether it’s HTTP, a message bus, raw TCP, web sockets, or anything else. This abstraction is the most important piece of infrastructure code.

In addition, microservices need a way to record behavior and errors. This means they need logging and a way to report their status. These are essentially the same, and a microservice shouldn’t concern itself with the details of log files or event reporting. In particular, microservices need to be able to fail fast, and fail loudly, so that the system and the team can take action quickly. A logging and reporting abstraction is also essential.[88]

Microservices also need an executive function. They should let service registries know about their existence so they can be managed. They should be able to accept external commands from administration and control functions in the system. Although communications and logging layers can often be provided by standalone libraries that you link into your services, the executive function depends on more-complex interactions with your custom administration and control functions. These layers must also play nicely with your deployment strategy and tooling. We’ll examine this in more detail in chapter 8.

[88] Using containers or serverless functions to deploy your microservices is a great way to get this type of tooling for free.

4.7 Summary

- The homogeneous nature of microservices makes them highly suitable as a fundamental unit of software construction. They’re practical units of functionality, planning, measuring, specification, and deployment. This characteristic arises from the fact that they’re uniform in size and complexity and are restricted to using messages to communicate with the outside world.
- A strict definition of the term *microservice* is too limiting. Rather, you generate ideas and expand the space of potential solutions by taking a more holistic viewpoint from a position of deeper understanding.
- Microservice architectures fall into two broad categories: synchronous (typically REST web services) and asynchronous (typically via a message queue). Neither is a full solution, and production systems are often hybrids.
- Monolithic architectures create three negative outcomes. They need more team coordination, causing management overhead; they suffer

from higher levels of technical debt, causing development speed to stall; and they’re high risk because deployments affect the entire system.

- The small size of microservices has positive outcomes. Estimation is more accurate, because microservices are mostly the same size; code is disposable, eliminating egocentric developer behaviors; and the system is dynamically configurable and so can more readily handle the unexpected.
- There are two types of code: business logic and structural libraries. They have very different needs. Microservices are for business logic, because they can handle the fuzzy, ever-changing requirements.
- To design a microservice system, start with requirements, express them as messages, and then group the messages into services. Keep all of this under control with a declarative model. Then think about how messages are handled by services: Synchronously or asynchronously? Observed or consumed?
- Microservices are natural software components. They are encapsulated and reusable, have well-defined interfaces, and, most important, can be composed together.