CHAPTER 1

What Spring AI Actually Does

IN THIS CHAPTER

What Spring AI is, and what it is not • The four pieces you will use • Installing a free model on your own machine • Creating the project • Your first program, on Ollama and then on OpenAI • What happens during one call • What to do when it does not work

Suppose your manager asks for something that sounds simple. Users should be able to type a question in plain English and get an answer drawn from the company's own documents.

You have built harder things than this. But when you sit down to start, nothing is where you expect it to be. The model lives behind someone else's HTTP API. Its replies come back as loose text, not as objects. Your documents are in a database that the model has never seen and cannot read. And every provider — OpenAI, Anthropic, Google, whoever your company picks next quarter — has its own SDK, its own request shape, its own idea of what a conversation is.

Spring AI exists for that gap. It is not a model. It does not make anything intelligent. It is the plumbing between the Java application you already know how to write and the AI models you do not want to hand-wire.

In this chapter you will install a free model on your own computer, write a Spring Boot program that asks it a question, and then point the same program at OpenAI by changing one line of configuration. No code changes.

1.1 A comparison you already know

Think about what JDBC did for databases.

Before JDBC, talking to MySQL and talking to Oracle were two different jobs. Each vendor shipped its own driver, its own connection handling, its own dialect of everything. Switching databases meant rewriting real code.

JDBC put one interface in front of all of them. You write against `Connection` and `PreparedStatement`. Underneath, a driver translates. Swapping Postgres for Oracle became a configuration change instead of a rewrite.

Spring AI applies exactly that idea to AI models.

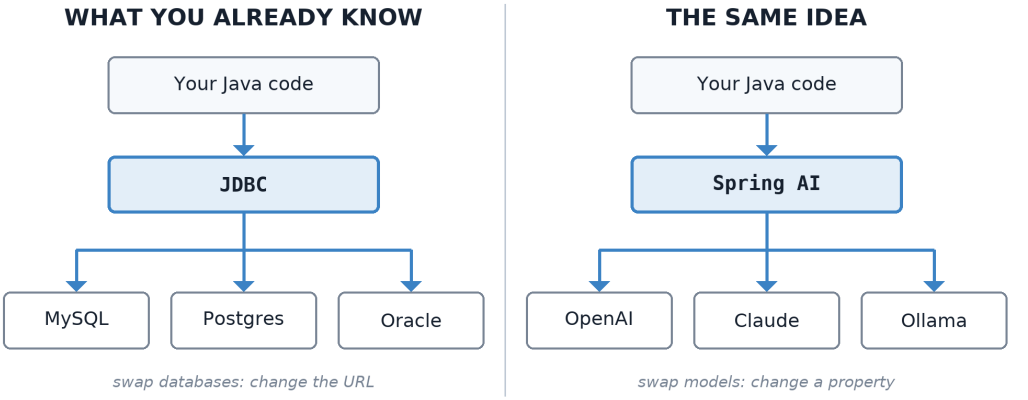

Figure 1.1 One API, many back ends. Spring AI does for AI models what JDBC did for databases.

You write against one API. Underneath, Spring AI translates to whichever provider you configured. Moving from OpenAI to Claude, or to a model running locally on your own machine through Ollama, becomes a change in `application.properties` rather than a change in your service classes. By the end of this chapter you will have done exactly that.

This matters more than it might seem. Model providers change their pricing, their rate limits, and their terms with very little notice. Teams that wired themselves directly into one SDK have found that expensive. Teams that went through an abstraction layer changed a property and moved on.

1.2 The pieces you will actually use

Spring AI has a long feature list. Most of it you can ignore at first. Four things do the real work.

ChatClient is the one you will use every day. You give it a question, it gives you an answer. If you have used `RestClient` or `WebClient`, it will feel familiar — the same fluent, chained style.

Embeddings turn text into a list of numbers. That sounds abstract and it is the single idea that makes everything else possible, so we will spend a

whole chapter on it later. For now: embeddings are how a computer measures whether two pieces of text mean similar things.

Vector stores hold those numbers so you can search them. Spring AI speaks to more than twenty of them — PostgreSQL with the pgvector extension, Redis, Chroma, Pinecone and others — through one common interface, the same way it handles model providers. If you already run Postgres, you already have a vector store and do not need new infrastructure.

Advisors sit between your call and the model and quietly modify it. One advisor remembers what was said earlier in the conversation. Another pulls relevant documents out of your vector store and attaches them to the question before it goes out. This is the piece that turns a generic model into something that answers using *your* data.

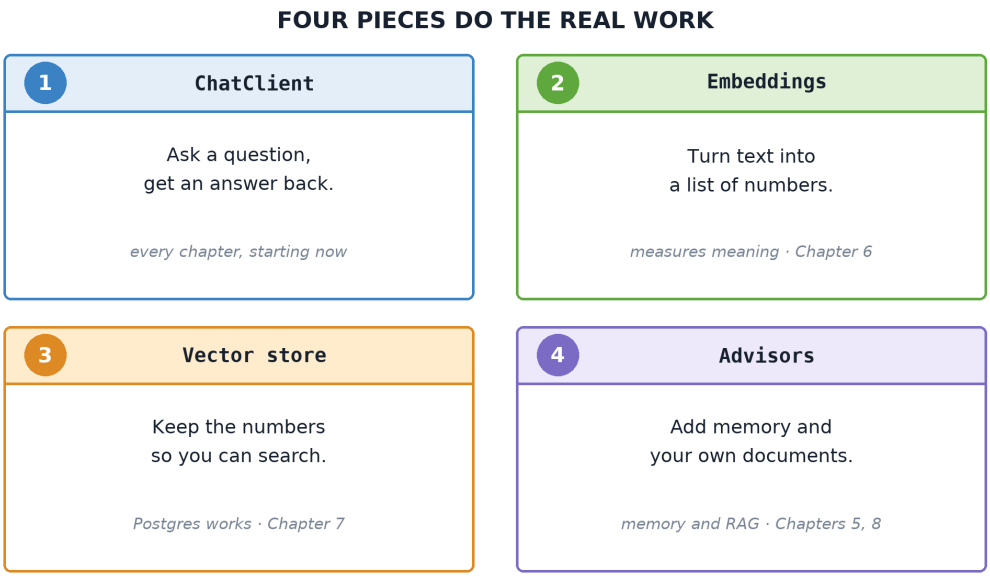

Figure 1.2 The four pieces, and the chapters where each one gets a project of its own.

That last pattern has a name you will see everywhere: Retrieval Augmented Generation, usually shortened to RAG. Strip away the jargon and it means: look things up first, then ask. You will build one in Chapter 8, and by then it will feel obvious.

1.3 Before you start

You need three things on your computer.

- **JDK 21.** Type `java -version` in a terminal. Spring AI 2.0 runs on Spring Boot 4, which needs at least Java 17. This book uses 21.
- **A Java IDE.** IntelliJ IDEA, VS Code with the Java extensions, or Eclipse. Anything that opens a Maven project.
- **Ollama.** A free program that runs AI models on your own computer. No account, no API key, no credit card.

You do not need to install Maven. Every project in this book includes the Maven wrapper, `mvnw`, which downloads the right Maven version the first time you use it.

The code for this book

Every program in this book is a complete Maven project, and all seventeen of them are here:

https://github.com/raj-ansh/spring-ai-examples

```text
git clone https://github.com/raj-ansh/
  spring-ai-examples.git
cd spring-ai-examples/ch01-hello
./mvnw spring-boot:run
```

One folder per project, named for its chapter: `ch01-hello`, `ch09-order-assistant`, and so on. Each opens in IntelliJ on its own (**File > Open**, pick the folder) and has a README saying what it does and what its output means.

The chapters build these projects from nothing, which is the point — you learn more typing the code than reading it. Use the repository when something will not compile, when you want to see the finished shape before starting, or when you would rather run a chapter than rebuild it. The code is MIT licensed, so anything useful can go straight into your own work.

Installing Ollama and a model

Download Ollama from `ollama.com` and run the installer. On Windows you can also type `winget install Ollama.Ollama`. Once installed, Ollama runs in the background and waits for requests on port 11434.

Ollama starts with no models. Download one:

```
Terminal

ollama pull qwen2.5:3b
```

`qwen2.5:3b` is a small model from Alibaba's Qwen team, a 1.9 GB download. Check that it arrived:

```
Terminal

ollama list
NAME          ID              SIZE
qwen2.5:3b    357c53fb659c    1.9 GB
```

(The real output also has a MODIFIED column, left out here.)

Every local run in this book was made on an ordinary laptop: an Intel Core i5-1235U with 16 GB of memory and no graphics card. If your machine is similar, the examples will run at the speeds you read about here.

Small local models are slower and less careful than paid cloud models, and this book shows that honestly, with real numbers, starting in Chapter 2. For learning how Spring AI works, they are ideal: free, private, and always there.

1.4 Creating the project

Spring Initializr, at `start.spring.io`, generates the skeleton of a Spring Boot project. Choose these settings:

Setting	Choose
Project	Maven
Language	Java
Spring Boot	4.1.1, or the newest 4.x
Group	`com.springaibook`
Artifact	`ch01-hello`
Packaging	Jar
Java	21

> **TWO SETTINGS TO CHECK**
> Initializr offered Java 17 by default when we generated this project. Change it to 21. And make sure the Spring Boot version starts with 4. Spring AI 2.0 cannot load under Spring Boot 3.x at all. Spring AI 2.0.x supports Boot 4.0.x and 4.1.x.

Now click **Add dependencies**. Type "Ollama" and add it. Then type "OpenAI" and add that too. Both sit in the AI section of the list, beside many other model providers and more than twenty vector stores.

You do not need **Spring Web**. This first program runs in a terminal, not a browser. Chapter 3 adds the web.

Click **Generate**, unzip the download and open the folder in your IDE.

What Initializr added

Open `pom.xml`. Three Spring AI entries matter:

pom.xml (the Spring AI parts)

```
org.springframework.ai:spring-ai-starter-model-ollama
org.springframework.ai:spring-ai-starter-model-openai
org.springframework.ai:spring-ai-bom
```

The two **starters** each bring one provider's client and the auto-configuration that sets it up. Notice that they have no version numbers. The version comes from the third entry, the **BOM** (bill of materials), which Initializr placed in `<dependencyManagement>`:

pom.xml

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.ai</groupId>
      <artifactId>spring-ai-bom</artifactId>
      <version>${spring-ai.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The property `spring-ai.version` is set to `2.0.1` near the top of the file. The BOM pins every Spring AI module to that one version, so they always match. When you add another Spring AI module in a later chapter, leave its version out too and let the BOM decide.

Everything here comes from Maven Central, the same place as the rest of your dependencies. You do not need to add any repositories.

1.5 Your first program

Enough theory. Open `src/main/resources/application.properties`. Initializr wrote one line into it, the application's name. Add the model settings:

application.properties

```
spring.application.name=ch01-hello

# Which provider answers: ollama or openai.
# This is the only line you change to switch.
spring.ai.model.chat=ollama

# Ollama: runs on your own machine, no API key.
spring.ai.ollama.chat.model=qwen2.5:3b

# OpenAI: reads your key from the OPENAI_API_KEY
# environment variable. Never paste the key here.
spring.ai.openai.api-key=${OPENAI_API_KEY}
spring.ai.openai.chat.model=gpt-4o-mini

# Keep the console to the program's own output.
spring.main.banner-mode=off
logging.level.root=WARN
```

Note the `${...}` syntax. That reads the key from an environment variable rather than hard-coding it in a file you might later commit. Keys have been leaked to public repositories more times than anyone wants to count. You do not need a key yet: while `spring.ai.model.chat` says `ollama`, the OpenAI settings are never used.

Now the whole program. Add this method to the `Ch01HelloApplication` class that Initializr generated:

Ch01HelloApplication.java

```
@Bean
```

Ch01HelloApplication.java

```
CommandLineRunner demo(ChatClient.Builder builder,
        ChatModel model) {
    return args -> {
        ChatClient client = builder.build();
        String question =
                "What is a Spring bean? One line.";
        String reply = client.prompt(question)
                .call()
                .content();
        System.out.println(reply);
        System.out.println("-- answered by "
                + model.getClass().getSimpleName());
    };
}
```

Your IDE will offer the imports: `ChatClient` and `ChatModel` from `org.springframework.ai`, `CommandLineRunner` and `Bean` from Spring. The complete file is in `code/ch01-hello`.

That is a working AI application. Fifteen lines, and two of them only close braces.

A `CommandLineRunner` runs once, just after the application starts. `builder.build()` gives you a `ChatClient`. `prompt(question)` says what to ask, `call()` sends it and waits for the reply, and `content()` hands you the answer as a `String`. The second parameter, `model`, is only there so the program can tell you which provider answered: Spring passes in whichever `ChatModel` it created, and the last line prints its class name.

Notice what you did not have to do. You did not build an HTTP client. You did not construct a JSON request body or read the provider's API reference to find out what shape it wants. You did not parse the response. Spring Boot's auto-configuration saw the starters on your classpath, read your settings, and handed you a ready-made `ChatClient.Builder`. This is the same auto-configuration you already rely on every time you add `spring-boot-starter-data-jpa` and get a working `EntityManager` without writing any setup code.

1.6 Running it

Run the program from the project folder. On macOS or Linux type `./mvnw spring-boot:run`, and on Windows `mvnw.cmd spring-boot:run`. The first run downloads Maven and every dependency, so give it a few minutes. After that it takes seconds.

This is the real output from `qwen2.5:3b`:

```
Console

A Spring bean is an injectable, configurable component.
-- answered by OllamaChatModel
```

It took 16 seconds from typing the command to seeing the answer, most of it Maven and Spring Boot starting up. Run the program again and the answer changes. Another run said:

```
Console

A Spring bean is an instance managed by the Spring
container.
-- answered by OllamaChatModel
```

Both are right. A model does not look answers up; it writes a new one every time. Keep that in mind for the rest of the book, because it is why Chapter 11 is about testing.

The line you must not forget

Take `spring.ai.model.chat=ollama` out of the properties file and run again. The program no longer starts:

```
Console (trimmed and wrapped)

APPLICATION FAILED TO START

Description:

Parameter 2 of method chatClientBuilder in
ChatClientAutoConfiguration required a single
bean, but 2 were found:
  - ollamaChatModel
  - openAiChatModel
```

```
Console (trimmed and wrapped)
Action:

Consider marking one of the beans as @Primary,
updating the consumer to accept multiple beans,
or using @Qualifier to identify the bean that
should be consumed
```

Read it as evidence, not as a disaster. You added two starters, so Spring Boot created two chat models, one for Ollama and one for OpenAI. `ChatClient.Builder` needs exactly one, and nobody said which. The suggestions under "Action" are general Spring advice and would work, but there is a simpler fix: `spring.ai.model.chat` names the provider, and Spring AI then creates only that one chat model. Put the line back.

Initializr does not add this line for you. Every project generated with two providers fails this way until you do.

1.7 Switching to OpenAI

Now the promise from section 1.1. For this part you need an OpenAI API key and some credit on the account. If you do not have one, read along; nothing later in the book requires it.

First put the key in an environment variable. In a macOS or Linux terminal:

```
Terminal
export OPENAI_API_KEY=sk-...
```

On Windows, `setx OPENAI_API_KEY "sk-..."` saves it for every new terminal. Close the terminal and open a new one afterwards, or the old one will not see it.

Then change one line in `application.properties`:

```
application.properties
1  spring.ai.model.chat=openai
```

Run the program again. Real output from `gpt-4o-mini`:

```
Console
A Spring bean is an object that is instantiated,
assembled, and managed by the Spring IoC (Inversion
of Control) container.
-- answered by OpenAiChatModel
```

The Java did not change. Not one line. Spring Boot read a different value, created an `OpenAiChatModel` instead of an `OllamaChatModel`, and handed your code a `ChatClient.Builder` that talks to OpenAI's servers.

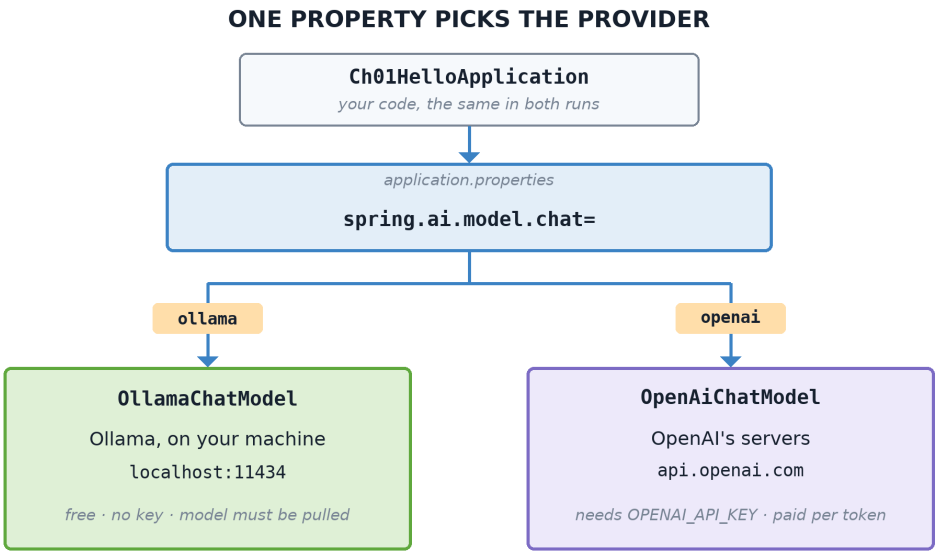

Figure 1.3 One property picks the provider. The code never changes.

You can also switch for a single run without editing the file. Build the jar, then pass the property on the command line:

```
Terminal
./mvnw package
java -jar target/ch01-hello-0.0.1-SNAPSHOT.jar
    --spring.ai.model.chat=openai
```

(Type the `java` command on one line.) Command-line values win over the properties file, so this run uses OpenAI and the next plain run goes back to Ollama. Chapter 3 goes further and uses both providers in one application.

Set the line back to `ollama` before you go on. Every project in this book starts on Ollama.

1.8 What actually happened

The call looks like one step. It is really six.

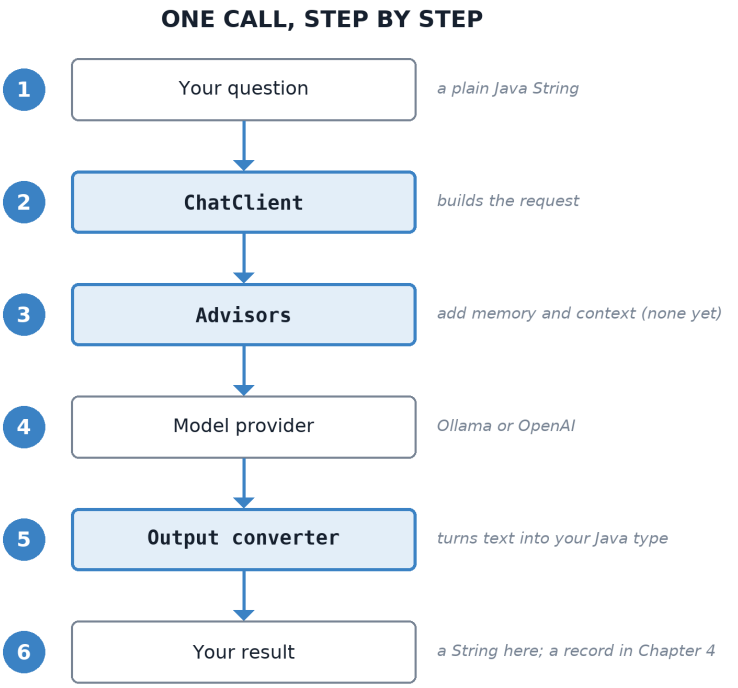

Figure 1.4 One call, step by step. The shaded steps are Spring AI; the provider is whichever one you configured.

Your question goes in as an ordinary string. `ChatClient` wraps it into a proper request. Any advisors you registered get a chance to add to it — conversation history, retrieved documents, extra instructions. The request goes out to whichever provider you configured. The reply comes back as text, and an output converter turns that text into whatever Java type you asked for.

In the example above you asked for `String`, so the converter had nothing to do. The interesting case is when you ask for something else.

1.9 Getting objects back, not text

This is where Spring AI stops being a thin HTTP wrapper and starts being useful.

Models return text. Text is awkward. If you want to do anything with an answer beyond printing it, you need it as an object.

In Spring AI you describe the object as an ordinary Java record, and replace `.content()` with `.entity(...)`, naming the record's class. Behind the scenes Spring AI inspects your record, describes its shape to the model, and parses the reply back into an instance. You get a typed object you can pass into a service, validate, or save. Not a string that looks like one.

That is what the project means when it talks about POJOs as the building blocks: you stay in ordinary Java, and the awkward translation happens out of sight. Chapter 4 builds an invoice parser this way, and shows what really happens on the way, including how often a small model gets it right.

> **A note on versions.** Spring AI moves quickly, and version 2.0 introduced breaking changes — tool execution in particular was reworked. Everything in this book targets **Spring AI 2.0.1 on Spring Boot 4.1.1 and Java 21**. If you are following an older tutorial and the code will not compile, a version mismatch is the first thing to check. All examples in this book are in the repository listed at the front, pinned to a known-good version.

1.10 When it doesn't work

Every problem below happened while we wrote this chapter. Each one has a message that tells you exactly what is wrong, once you know how to read it.

"required a single bean, but 2 were found". Both starters are on the classpath and `spring.ai.model.chat` is missing. Add it, as in section 1.6. Setting it to an empty value does not help: then Spring AI creates no chat model at all, and the error says a `ChatModel` bean "could not be found".

The program sits there, printing "Retry error". Ollama is not running, or the program is looking on the wrong port:

```
Console (trimmed and wrapped)

WARN ... Retry error. Retry count:1
ResourceAccessException: I/O error on POST
request for "http://localhost:11999/api/chat":
Connection refused
```

(We pointed the program at an unused port on purpose, to see this.) Spring AI retries a failed call up to ten times, waiting 2 seconds, then 10, then 50, five times longer each time. In our test the program was still waiting seven minutes later. Start Ollama and run again. While you are developing, you can add `spring.ai.retry.max-attempts=3` to the properties file, and it gives up after about a minute instead.

"model 'llama3.2:1b' not found". Ollama is running, but the model named in `spring.ai.ollama.chat.model` has not been downloaded:

```
Console (trimmed and wrapped)

NonTransientAiException: HTTP 404 -
{"error":"model 'llama3.2:1b' not found"}
```

This one fails at once, with no retries, because trying again cannot help. Run `ollama pull` with the name from your properties, or fix the name.

The build asks the model a question. Initializr also generated a test, `contextLoads`, which starts the whole application. Starting the application runs your `CommandLineRunner`, so `./mvnw package` calls the model too, and needs Ollama running, or a valid key if you switched to OpenAI.

The OpenAI key is not found in your IDE. IDEs do not always see environment variables set after they started. Restart VS Code completely after setting the key. In IntelliJ IDEA, add the variable to the run configuration itself.

HTTP 429 from OpenAI on the very first call. A 429 that mentions your quota means the account has no credit. It is a billing problem, not a code problem. In fact the stack trace proves your wiring is right: the call reached OpenAI and OpenAI answered.

A dependency called `spring-ai-openai-spring-boot-starter` will not resolve. That is the starter's old name, still all over published tutorials. Since Spring AI 1.0 it is `spring-ai-starter-model-openai`, and the old name is not in the 2.0 BOM.

Code from a tutorial will not compile. Check its Spring AI version. Anything written for 1.x on Spring Boot 3 differs from this book in places, and Spring AI 2.0 will not run on Spring Boot 3 at all.

1.11 What you know now

- Spring AI is an adapter layer, not a model. It gives you one API across many providers, the same way JDBC gave you one API across many databases.
- `ChatClient` is your entry point. Advisors are how you inject memory and your own documents. Output converters are how you get Java objects instead of loose text.
- Initializr gives you the starters and the BOM. You add `spring.ai.model.chat` and the model names yourself.
- Switching from Ollama to OpenAI is one property. The code does not change.
- The same question gets a different answer each time.
- Error messages are evidence. A stack trace that reaches the provider proves the wiring works.

You have run a working AI application on two providers, and you understand every line in it.

In the next chapter we look at what a prompt really is — because the model does not see the neat question you typed. It sees something rather different, and knowing what it sees explains a great deal of behaviour that otherwise looks like magic.

1.12 Try it yourself

1. Change the question to one about your own work, and ask for three lines instead of one. Does the model keep to three?
2. Run the program three times on Ollama and write down the answers. How many different ones do you get?
3. Stop Ollama, add `spring.ai.retry.max-attempts=3` to the properties file and run. Read the timestamps on the "Retry error" lines. How long did each wait last?