Chapter 12. Pipeline Architecture Style

One of the fundamental styles in software architecture is the *pipeline* architecture (also known as the *pipes and filters* architecture). As soon as developers and architects decided to split functionality into discrete parts, this style of architecture followed. Most developers know this architecture as the underlying principle behind Unix terminal shell languages, such as Bash and Zsh.

Developers in many functional programming languages will see parallels between language constructs and elements of this architecture. In fact, many tools that use the MapReduce programming model follow this basic topology. While the examples in this chapter show low-level implementations of the pipeline architecture style, it can also be used for higher-level business applications.

Topology

The topology of the pipeline architecture consists of two main component types, pipes and filters. *Filters* contain the system functionality and perform a specific business function, and *pipes* transfer data to the next filter (or filters) in the chain. They coordinate in a specific fashion, with pipes forming one-way, usually point-to-point communication between filters, as illustrated in Figure 12-1.

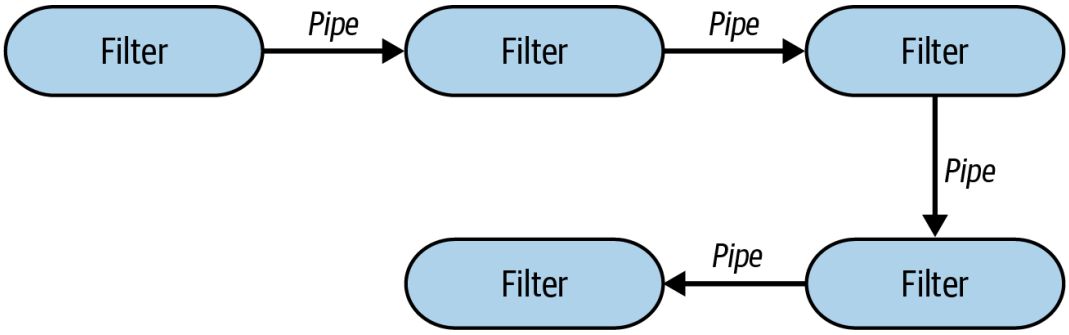

Figure 12-1. Basic topology for pipeline architecture

The isomorphic "shape" of the pipeline architecture is thus a *single deployment unit, with functionality contained within filters connected by unidirectional pipes*.

Style Specifics

While most implementations of the pipeline architecture are monolithic, it is possible to deploy each filter (or a set of filters) as a service, creating a distributed architecture with synchronous or asynchronous remote calls to each service. Whatever its deployment topology, the architecture consists of only two architectural components, filters and pipes, which we describe in detail in the following sections.

Filters

Filters are self-contained pieces of functionality that are independent from other filters. They are generally stateless, and should perform one task only. Composite tasks are typically handled by a sequence of filters rather than a single one.

Because filters can be implemented by more than one class file, they are considered *components* of the architecture (see Chapter 8). Even if a filter is simple and only implemented as a single class file, it's still a component.

There are four types of filters in the pipeline architecture style:

Producer

> The starting point of a process, producer filters are outbound only. They are sometimes called the *source*. A user interface and an external request to the system are both examples of producer filters.

Transformer

> Transformer filters accept input, optionally perform a transformation on some or all of the data, then forward the data to the outbound pipe. Functional-programming advocates will recognize this feature as *map*. Transformer filters might, for example, enhance data, transform data, or perform some sort of calculation.

Tester

> Tester filters accept input, test it according to one or more criteria, and then optionally produce output based on the test. Functional programmers will recognize this as similar to *reduce*. A tester filter might check that all data is valid and entered correctly, or act as a switch to determine if processing should move forward (for example, "don't forward the data to the next filter if the order amount is less than five dollars").

Consumer

> The termination point for the pipeline flow, consumer filters sometimes persist the final result of the pipeline process to a database or display the final results on a UI screen.

The unidirectional nature and simplicity of pipes and filters encourage compositional reuse. Many developers have discovered this ability using shells. A famous story from the blog "More Shell, Less Egg" illustrates just how powerful these abstractions are. Donald Knuth was asked to write a program to solve this text-handling problem: read a file of text, determine the *n* most frequently used words, and print a list of those words sorted by frequency. He wrote a program consisting of more than 10 pages of Pascal, designing (and documenting) a new algorithm along the way. Then Doug McIlroy demonstrated a shell script small enough to easily fit within a social media post that solved the problem elegantly:

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Even the designers of Unix shells are often surprised at the inventive uses developers find for their simple but powerful composite abstractions.

Pipes

Pipes, in this architecture, form the communication channel between filters. Each pipe is typically unidirectional and point-to-point, accepting input from one source and directing output to another. Their payload can be any data format, but architects typically favor smaller amounts of data to enable high performance.

If a filter (or group of filters) is deployed as a separate service in a distributed fashion, the pipes issue a unidirectional remote call using REST, messaging, streaming, or some other remote communication protocol. Whether their deployment topology is monolithic or distributed, pipes can be either synchronous or asynchronous. In monolithic deployments, architects use threads or embedded messaging for asynchronous communication to a filter.

Data Topologies

Because most pipeline architectures are deployed as monoliths, they are associated with a single monolithic database. However, this is not always the case. Database topology can vary significantly with this architectural style, from a single database to one database per filter.

The example in Figure 12-2 shows a pipeline architecture for a typical continuous fitness function (architectural test) running in a production environment that analyzes a specific operational characteristic (such as responsiveness or scalability). Notice that the Capture Raw Data filter loads a separate database containing raw data. This filter sends the raw data through a pipe to the Time Series Selector filter, which reads configuration information (such as the period of time being analyzed) from a separate database. The transformed data is then sent through another pipe to the Trend Analyzer filter, which analyzes the data and stores those analytics in a separate database. This filter then sends the analytics data through a final pipe to the Graphing Tool filter, which ends the pipeline by generating a graphical report of the data.

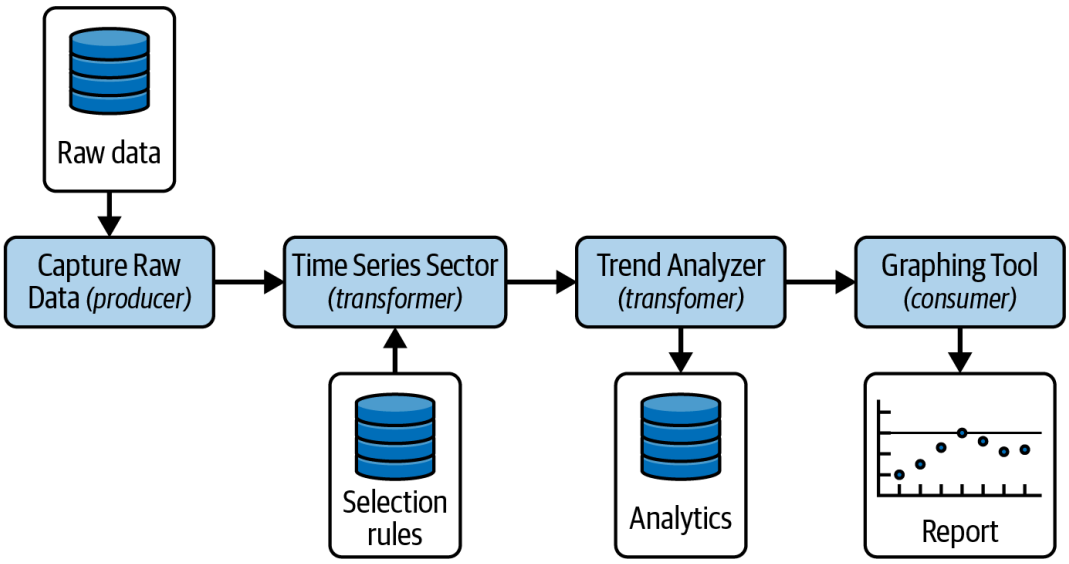

Figure 12-2. Pipeline architectures can have a single database or many

Cloud Considerations

The pipeline architecture style is well suited for cloud-based deployments due to its high level of modularity and separate filter types. Because most pipeline architectures are not overly complex and extensive, they work well being deployed as monolithic architectures, where all filters are deployed in the same deployment unit.

However, filters also work well in cloud environments as distributed functions. In AWS, the pipeline architecture can be deployed as AWS Step Functions, where each filter is deployed as a separate lambda in the workflow. AWS Step Functions offer two workflows: *Standard*, in which each step is executed exactly once, and *Express*, where each step can execute more than once. The pipeline architecture style works with both. The following code illustrates the continuous fitness function example in Figure 12-2 as a typical cloud deployment for the pipeline architecture, implemented as an AWS Step Function:

```
{
  "Comment": "Measure and analyze scalability trends.",
  "StartAt": "Capture Raw Data",
  "States": {
    "Capture Raw Data": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:raw_data_capture",
    "Next": "Time Series Selector"
   },
    "Time Series Selector": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:time_series_selector",
    "Next": "Trend Analyzer"
   },
    "Trend Analyzer": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:trend_analyzer",
    "Next": "Graphing Tool"
    },
    "Graphing Tool": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:region:account_id:function:graphing_tool",
    "End": true
    }
  }
}
```

This example isn't the only way to use the pipeline architecture in cloud environments. Filters can also be deployed as serverless functions, as containerized functions, or even as a single service containing all four filter components in a monolithic deployment.

Common Risks

The primary goal behind the pipeline architecture is to separate functionality into single-purpose filters, where each filter performs *one* specific action on the data and then hands it off to another filter for further processing. As such, one of its most common risks is overloading filters with too much responsibility. Good governance, which we cover in the next section, helps teams mitigate this risk by identifying the purpose behind each filter component.

Another common risk with this architecture style is introducing bidirectional communication between filters. Pipes are meant to be *unidirectional only*,

providing a clear separation of concerns between filters to avoid collaboration between them. If bidirectional communication turns out to be necessary, this is a good indication that the pipeline architecture might not be the right style to use, or that the filters are too complex; functionality isn't demarcated correctly.

Handling error conditions is another complication that can introduce significant risk within this architectural style. If an error occurs within a pipeline, it is often difficult to determine how to properly exit the pipeline and recover once the pipeline is started. For this reason, it's important for architects to determine any possible fatal error conditions within the pipeline before defining the architecture.

The last risk area is managing the contracts between filters. Each pipe has a contract representing the data (and possibly the corresponding types) that it sends to the next filter. Changing a contract between filters requires strict governance and testing to ensure that other filters receiving the contract don't break.

Governance

Governance for general operational characteristics—such as responsiveness, scalability, availability, and so on—is specific to each individual use case and can vary greatly. However, from a *structural* standpoint, architects use the role and responsibility of each filter type to govern pipeline architectures.

Each of the four basic filter types (producer, transformer, tester, and consumer) performs a specific role. However, it's all too easy for developers to overload filters with too much responsibility, turning the pipeline architecture into an unstructured monolith.

It is difficult to write an automated fitness function to govern whether a producer filter is actually the starting point of the pipeline, or whether a tester filter is actually performing a conditional check to determine whether to continue with the flow or terminate it. Governance techniques help architects guide the development team in adhering to each filter type's role and the responsibility of a specific filter type.

One such technique is to leverage *tags*. (In the Java language, tags are implemented as *annotations*, and in C# as *custom attributes*.) Tags don't actually perform any function; they programmatically provide metadata about the component or service. Utilizing tags in the starting class of the filter component tells developers about the type of filter they are creating or modifying, helping prevent them from giving that filter too much responsibility or performing a function not associated with its type.

To illustrate this technique, consider the following source code that defines tags for the four basic filter types. Here is the Java tag annotation definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Filter {
   public FilterType[] value();

   public enum FilterType {
      PRODUCER,
      TESTER,
      TRANSFORMER,
      CONSUMER
   }
}
```

And here is the C# tag custom attribute definition:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class Filter : System.Attribute {

   public FilterType[] filterType;

   public enum FilterType {
      PRODUCER,
      TESTER,
      TRANSFORMER,
      CONSUMER
   };
}
```

Since filters are essentially an architectural component, they can be implemented through multiple class files. For example, in our continuous fitness function example in Figure 12-2, the Trend Analyzer filter is fairly complex and could have multiple class files. Because of this, we'll define one additional tag to identify the class that represents the filter's entry point, so we can attach other tags to it.

Here is the Java entry point tag definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FilterEntrypoint {}
```

And the C# entry point tag definition:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
class FilterEntrypoint : System.Attribute {}
```

Now the developers can add the `FilterType` tag to the `Entrypoint` class, identifying its type and its corresponding role in the architecture. For example, the following annotations identify the type and role of the Trend Analyzer transformation filter. In Java:

```
@FilterEntrypoint
@Filter(FilterType.TRANSFORMER)
public class TrendAnalyzerFilter {
   ...
}
```

And in C#:

```
[FilterEntrypoint]
[Filter(FilterType.TRANSFORMER)]
class TrendAnalyzerFilter {
   ...
}
```

While this technique might not stop every developer from making a transformer filter type perform testing logic (which should be done by a tester filter), at least it provides additional context.

Team Topology Considerations

Unlike some of the architectural styles described in this book, the pipeline architecture style is generally independent of team topologies and will work with any team configuration:

Stream-aligned teams

Because the pipeline architecture is typically small and self-contained and represents a single journey or flow through the system, it works well with stream-aligned teams. With this team topology, teams generally own the flow through the system from beginning to end, nicely matching the shape of the pipeline architecture.

Enabling teams

Because the pipeline architecture is highly modular and separated by technical concerns, it pairs well with enabling team topologies. Specialists and cross-cutting team members can make suggestions and perform experiments by introducing additional filters within the pipeline, without affecting the rest of the flow. For example, in our continuous fitness function example, an enabling team might add a transformation filter after the Time Series Selector filter to do some alternative trend analysis. This new filter would use the same data as the existing Trend Analyzer filter without disrupting the normal pipeline flow.

Complicated-subsystem teams

Because each filter performs a very specific task, the pipeline architecture works well with the complicated-subsystem team topology. Different team members can focus on complicated filter processing independently from one another (and from other filters). The unidirectional handoffs involved in this architectural style allow members of complicated-subsystem teams to focus narrowly on the complexity contained within their specific filter processing.

Platform teams

Platform teams working on a pipeline architecture can leverage its high degree of modularity by utilizing common tools, services, APIs, and tasks.

Style Characteristics

A one-star rating in the characteristics ratings table (Figure 12-3) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The characteristics contained in the scorecard are described and defined in Chapter 4.

The pipeline architecture style is a *technically partitioned* architecture because its application logic is separated into filter types. Also, because the pipeline architecture is usually implemented as a monolithic deployment, its architectural quantum is always 1.

	Architectural characteristic	Star rating
	Overall cost	$
Structural	Partitioning type	Technical
Structural	Number of quanta	1
Structural	Simplicity	★★★★
Structural	Modularity	★★
Engineering	Maintainability	★★
Engineering	Testability	★★★
Engineering	Deployability	★★
Engineering	Evolvability	★★★
Operational	Responsiveness	★★★
Operational	Scalability	★
Operational	Elasticity	★
Operational	Fault tolerance	★

Figure 12-3. Pipeline architecture characteristics ratings

Overall cost, simplicity, and modularity are the primary strengths of the pipeline architecture style. Being monolithic, pipeline architectures don't have the complexities associated with distributed architecture styles—they're simple and easy to understand, and are relatively low-cost to build and maintain. Architectural modularity is achieved through separating concerns between the various filter types and transformers: any filter can be modified or replaced without affecting the other filters. For instance, in the Kafka example illustrated in Figure 12-4, the `Duration Calculator` can be modified to change the duration calculation without changing any other filter.

Deployability and testability, while only average, rate slightly higher than in the layered architecture due to the level of modularity filters can achieve. However, pipeline architectures are still *typically* monolithic, so their downsides include ceremony, risk, frequency of deployment, and completeness of testing.

Elasticity and scalability rate very low (one star) in the pipeline architecture, primarily due to monolithic deployments. Implementing this architecture style as a distributed architecture using asynchronous communication can significantly improve these characteristics, but the trade-off is a blow to overall cost and simplicity.

Pipeline architectures don't support fault tolerance because they are typically deployed as monolithic systems. If one small part of a pipeline architecture causes

an out-of-memory condition to occur, the entire application unit is impacted and crashes. Furthermore, overall availability is affected by the high mean time to recovery (MTTR) common in most monolithic applications, with startup times measured in minutes. As with elasticity and scalability, implementing this architecture style as a distributed architecture using asynchronous communication can significantly improve fault tolerance, again paying the price with cost and complexity.

As we've mentioned, most of the low-scoring operational characteristics can be raised by making this a distributed architecture with asynchronous communication, where each filter is a separate deployment unit and the pipes are remote calls. However, doing so will negatively affect other attributes such as simplicity and cost, illustrating one of the classic trade-offs of software architecture.

When to Use

The pipeline architecture is suitable for systems of any complexity with distinct, ordered, and deterministic one-way processing steps. This style's simplicity also makes it well suited for situations involving tight time frames and budget constraints.

When Not to Use

Because of the monolithic nature of this architectural style, it's not a good fit for systems that need high scalability, elasticity, and fault tolerance. However, using a distributed architecture approach will help mitigate these concerns.

The pipeline architecture style is also ill-suited for scenarios involving back-and-forth communication between filters, since the pipes are meant to be unidirectional only. While it's possible to use this architecture style for nondeterministic workflows by leveraging lots of tester filters throughout the pipeline, we don't recommend doing so. It overcomplicates this relatively simple architecture style and can negatively affect maintainability, testability, deployability, and consequently overall reliability. Event-driven architecture (see Chapter 15) is much better suited for situations involving nondeterministic workflows.

The pipeline architecture style appears in a variety of applications, especially with tasks that facilitate simple, one-way processing. For example, many electronic data interchange (EDI) tools use this pattern, building transformations from one document type to another using pipes and filters. Database extract, transform, and load (ETL) tools leverage pipeline architectures to modify data and flow it from one database or data source to another. Orchestrators and mediators such as Apache Camel use the pipeline architecture to pass information from one step in a business process to another.

Examples and Use Cases

To illustrate how the pipeline architecture can be used, consider the following example, where service telemetry information is sent from services via streaming to Apache Kafka (Figure 12-4).

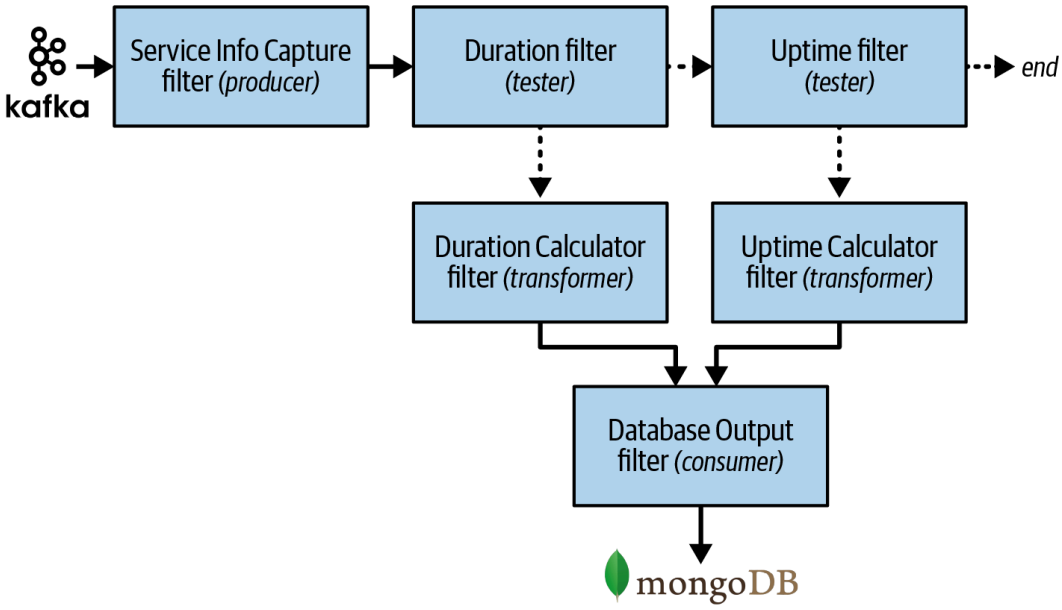

Figure 12-4. Pipeline architecture example

The system depicted in Figure 12-4 uses the pipeline architecture style to process different kinds of data streamed to Kafka. The Service Info Capture filter (producer filter) subscribes to the Kafka topic and receives service information. It then sends this captured data to a tester filter called the Duration filter to determine whether it's related to the duration (in milliseconds) of the service request.

Notice the separation of concerns between the filters; the Service Info Capture filter is only concerned with how to connect to a Kafka topic and receive streaming data, whereas the Duration filter is only concerned with qualifying the data and determining whether to route it to the next pipe. If the data is related to the service request duration, it is passed to the Duration Calculator transformer filter; otherwise, the data is passed to the Uptime tester filter to check if it's related to uptime metrics. If not, then the pipeline ends—the data is of no interest to this particular processing flow. If it is relevant to uptime metrics, the Uptime tester filter passes it to the `Uptime Calculator`, which uses it to calculate the uptime metrics. The `Uptime Calculator` then passes the modified data to the Database Output consumer, which persists it in a MongoDB database.

This example shows how extensible the pipeline architecture is. For example, in Figure 12-4, a new tester filter could easily be added after the Uptime filter to send the data to be used in a new metric, such as for database connection wait time.

The pipeline architectural style, while typically a monolithic architecture, nevertheless demonstrates a good level of modularity thanks to the use of technically partitioned filters. It's a good fit for situations involving a workflow-based, step-by-step approach to processing data within a system.

Before we move on to more complicated distributed architectures, we have one more monolithic architecture to describe in the next chapter that also supports a good level of modularity through the use of plug-in components: the *microkernel architecture*.