O'REILLY®

Reactive Systems Architecture

DESIGNING AND IMPLEMENTING AN ENTIRE DISTRIBUTED SYSTEM

Jan Machacek, Martin Zapletal,
Michal Janousek & Anirvan Chakraborty

JOIN THE CAKE SOLUTIONS TEAM!

As an engineer at Cake Solutions, you will stay at the forefront of technology. You will be encouraged to share your thoughts through blogs, open-source contributions, conferences, and much more; becoming an advocate for a principled approach to software engineering.

Why Cake Solutions:

- We deliver great solutions to an incredible client list
- We build highly distributed systems
- We are renowned for our work with Scala, Spark, Mesos, Akka, Cassandra and Kafka
- We jump ship to Python, Golang, C++ and many others where appropriate
- We build systems that ingest and process data from the edge of IoT
- We research and deliver production-grade ML systems
- We embrace the DevOps culture

Visit our job portal at
http://www.cakesolutions.net/jobs

CAKESOLUTIONS

Reactive Systems Architecture

Designing and Implementing an Entire Distributed System

This Preview Edition of *Reactive Systems Architecture*, Chapter 7, is a work in progress. The final book is currently scheduled for release in August 2017 and will be available at *oreilly.com* and other retailers once it is published.

Jan Machacek, Martin Zapletal, Michal Janousek, and Anirvan Chakraborty

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Reactive Systems Architecture
by Jan Machacek, Martin Zapletal, Michal Janousek, and Anirvan Chakraborty

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian Foster
Production Editor: Nicholas Adams
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2017: First Edition

Revision History for the First Edition
2017-03-13: First Preview Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491980712* for release details.

978-1-491-98662-2

[LSI]

Table of Contents

CHAPTER 1

Image processing system

The system we are going to describe in this chapter accepts images and produces structured messages that describe the content of the image. Once the image is ingested, the system uses several independent microservices, each performing a specific computer vision task and producing a response specific to its purpose. The messages are delivered to the clients of the system. The microservices are containerised using Docker, most of the microservices are implemented using Scala[scala], the computer vision ones are implemented in C++ and CUDA. The event journals and offset databases is running in Redis containers. Finally, the messaging infrastructure (Apache Kafka) is running outside any container. All components are managed in a DC/OS distributed kernel and scheduler; Consul[consul] provides the service discovery services; Sumologic[sumologic] logging and metrics; finally Pingdom[pingdom] provides customer-facing service availability checks.

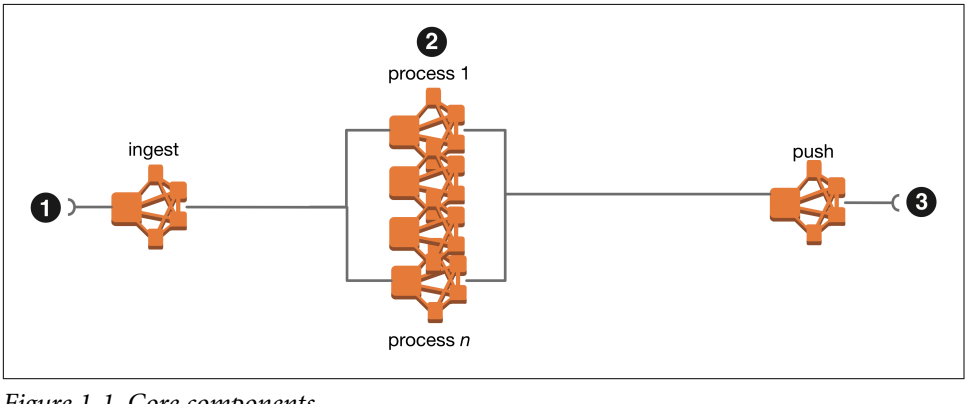

Figure 1-1. Core components

Let's take a look at the key points in Figure 1-1, starting with the external inputs and outputs:

❶ Clients send their requests to the `ingestion` service; the response is only a confirmation of receipt, it is not the result of processing the image.

❷ The *process* microservices perform the computer vision tasks on the inputs. The microservices emit zero or more messages on the output queue.

❸ The `push` microservice delivers each message from the vision microservices to the clients using ordinary HTTP POSTs to the client's public-facing endpoints.

Architectural concerns

Before we start discussing the implementation of this system, we need to consider what information we will handle, and how we're going to route it through our system. Moreover, we need to guarantee that the system will not lose a message, which means that we will need to consider the implications of at-least-once delivery semantics in a distributed system. Because we have a distributed system, we need to architect the system so that we reduce the impact of the inevitable failures.

Let's begin by adding a requirement for a summary service, which makes integration easier and brings additional value to our clients by combining the output of the vision microservices—and using our knowledge of the vision processes—produce useful high-level summaries of multiple ingested messages. It would be tempting to have the summary service be at the centre of the system: it receives the requests, and *calls* other components to perform their tasks. Along the same lines, it would also be easy to think that there are certain services which simply *must* be available. For example, without the authentication and authorisation services, a system simply cannot process requests. (See Figure 1-2.)

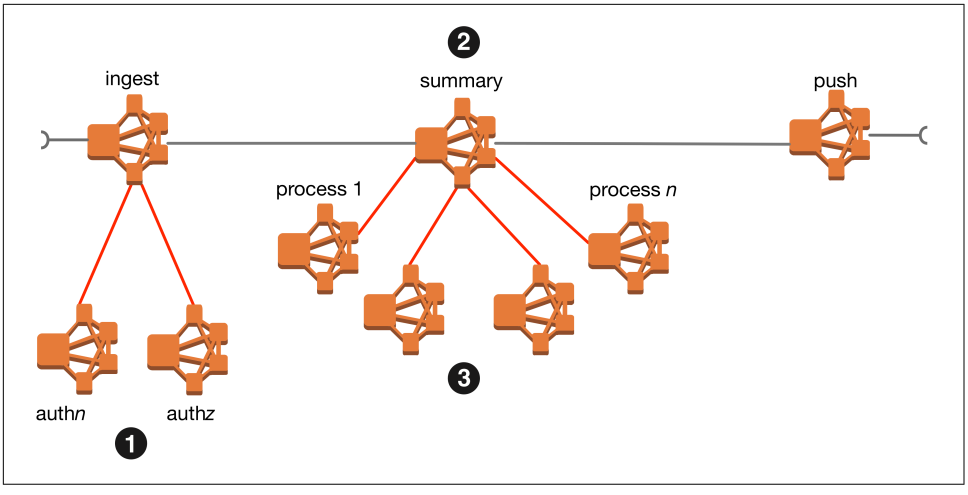

Figure 1-2. Orchestrated architecture

Externally, the system looks the same; but internally, it introduces complex flow of messages and the inevitable time-outs in the interaction between <2> and <3>. Architecture that attempts to implement request-complete response semantics in a distributed messaging environment often leads to complex state machines—here inside the `summary` service—because it must handle the communication with its dependant services <3> *as well as* the state it needs to compute its result. If the `summary` needs to shard its domain across multiple nodes, we end up with the `summary` cluster. Clustered services bring even more complexity, because they need to contain state that describes the topology of the cluster. The more state the service maintains and the more the state is spread, the more difficult it's going to be to maintain consistency of that state. This is particularly important when the topology of the cluster changes: either as a result of individual node failures, network partitions, or even planned deployment. We avoid designing our system with a central orchestrating component: such a component will become the monolith we are trying to avoid in the first place.

Another architectural concern is daisy-chaining of services, where the flow of messages looks like a sequence of function calls, particularly if the services in the chain make decision about the subsequent processing flow. The diagram in Figure 1-3 shows such daisy-chaining.

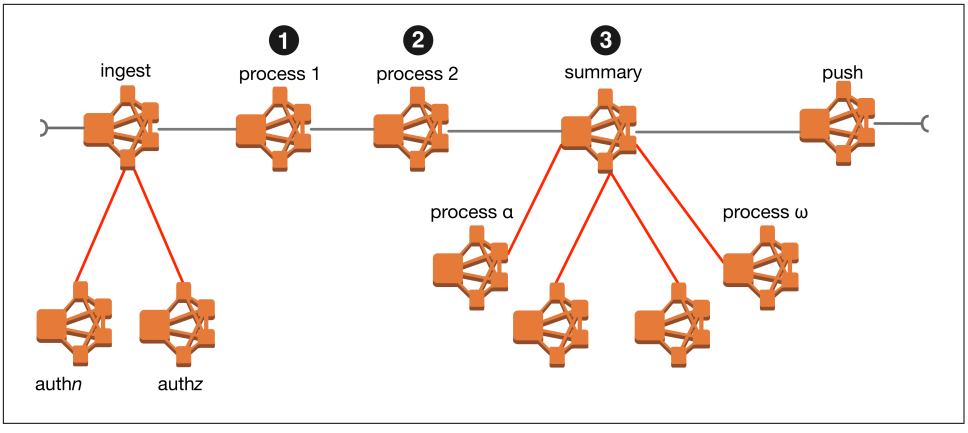

Figure 1-3. Daisy-chaining services

In the scope of image processing, imagine that the service <1> performs image conversion to some canonical format and resolution, and <2> performs image quality and rough content checks; only if the conversion and image quality checks succeed do we proceed to deal with the input. The flow of the messages through the system can be described in pseudo-code in Example 1-1.

Example 1-1. Daisy-chaining services

```
byte[] input = ...;
ServiceOneOutput so1 = serviceOne(input);
if (so1.succeeded) {
  ServiceTwoOutput so2 = serviceTwo(so1.output);
  if (so2.succeeded) {
    ServiceThreeOutput so3 = serviceThree(so2.output);
    ...
  }
}
```

Remember though that `serviceOne`, `serviceTwo`, and `serviceThree` are services that live in their own contexts, isolated from each other by a network connection. This flow, when implemented using network calls is inefficient, but that's not the biggest problem. The biggest problem is that `serviceOne` needs to convert the input image into the format that is optimal for the downstream services. Similarly, `serviceTwo` needs to be *absolutely certain* that if it rejects the input, the subsequent processing wound indeed fail. Let's now improve one of the downstream services—perhaps the OCR service can now successfully extract text from images of much lower quality. Unfortunately, we will not be able to see the impact of the improvement unless we also change the quality check service. (The scenario is very similar to a scenario

where the downstream services can now use high-resolution images to perform some fine detail processing; a resolution that the conversion service always downsamples.)

To solve these problems, we must make allow the services to be as loosely-coupled as possible; and to allow each microservice to completely *own* the state it is responsible for managing, but to keep this area of responsibility sharply defined and as coherent as possible. To enable loose-coupling, do not discard information if possible: it is always easier to compose services if the services *enrich* the incoming messages. Do not create *x* with *y*-like services (ingestion *with* conversion) unless there is insurmountable technical reason (e.g. the ingestion and conversion microservice has to work on a special hardware component). Wherever possible, steer away from request-required response—specifically request-required *complete* response—messaging patterns: this can lead to brittle systems, because the service being called has to be available and has to complete the processing in very short period of time. For authorisation and authentication, we should use token-based approaches, where the token is all that any service needs for authorisation: there is no need to make a (synchronous request-required complete response) call to the authorisation service. This leads us to architecture in Figure 1-4.

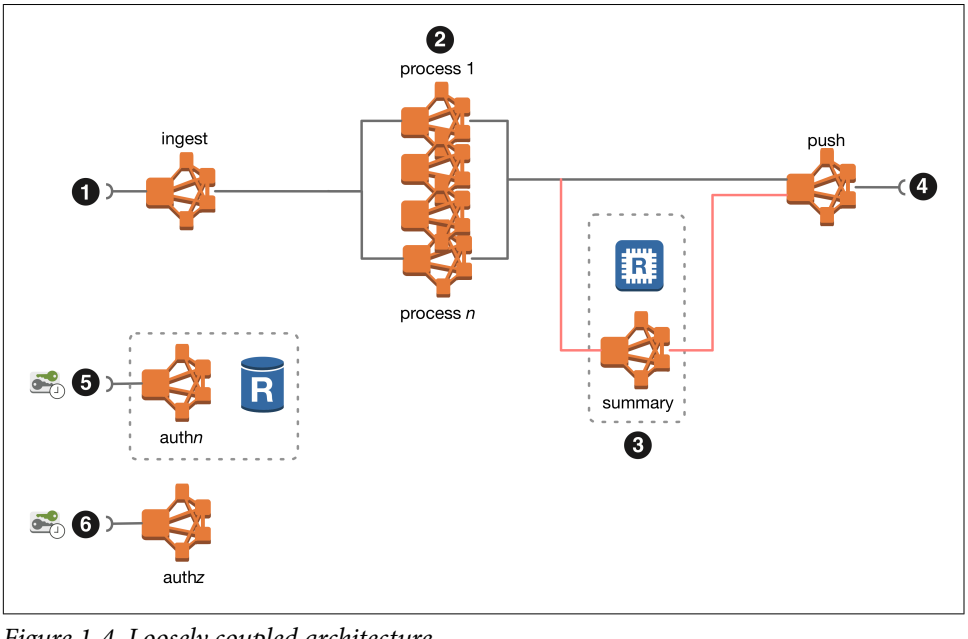

Figure 1-4. Loosely coupled architecture

Now that we have a world where the services communicate with each other using asynchronous messaging (a service may consume or produce messages at any time), we have to carefully consider how we're going to route the messages in our system. We want a message delivery mechanism that allows us to publish a message (to a

known "location"), and to subscribe to receive messages from other known "locations". This can be achieved with REST: the location is the endpoint (typically behind some kind of service gateway), and the subscription can be a simple web hook, where a microservice maintains a list of endpoints to which it will make the appropriate REST calls. This approach is missing the ability to easily de-couple the endpoints in time. A complete message broker achieves the same asynchronous message delivery; some brokers add persistence and journalling, which allows us to treat the messaging infrastructure as the *event store* for the microservices. This allows us to achieve at-least-once delivery semantics with little additional overhead. The ingestion and push services are still there; so are the vision services.

❶ Clients that do not need to receive any response (apart from confirmation of receipt—think HTTP 200) send their requests to the `ingestion` service. It accepts the request, performs validation and initial pre-processing; as a result of the initial processing, it places on the message queue for processing.

❷ The processing services form a consumer group (containing multiple instances of the same microservice) for each computer vision task; the broker delivers the one message to one thread in each consumer group. The vision microservices place the result in one or more messages on the results queue.

❸ The `summary` service aggregates the vision result messages to derive deeper results from the results of the vision components. Imagine being able to track a particular object over multiple requests, identify the object being tracked, etc.

❹ The `push` micoservice delivers each message from the vision microservices to the clients using ordinary HTTP POSTs; the clients are expected to implement endpoints that can process the results of the vision microservices; this endpoint must be able to handle the traffic that this system generates and the logic behind the client endpoint must be able to handle correlation and de-duplication of the received messages.

❺ The authentication service manages credentials for clients to use the system; the clients are mostly *other systems*, mobile applications and smart devices that need to be identified and allowed ask for authorisation to use the system's services.

❻ The authorisation service turns tokens issued by the authentication service into authorisation tokens, which contain the details of the resources that the bearer of that token can use. The token is checked not just at the `ingestion` service, but throughout the system.

Before we turn to the important details of the implementation, let's discuss cross-service concerns.

Protocols

It is crucial for any [distributed] system to precisely define the protocols that the components use to communicate. Having precise protocols allows us to be precise about the compatibility of the different microservices and to precisely explain to our clients the messages that the system produces. Moreover, we can use these protocol definitions to accelerate the implementation of the different microservices: if each microservice knows the protocol, it can trivially validate its inputs and it can *generate* synthetic outputs. This allows us to build a *walking skeleton*: a complete implementation of all the microservices and message paths, without having to spend the time to implement the functionality of each of the microservices.

A good protocol definition gives us the flexibility to maintain compatibility even as we add and remove fields. There are a number of protocol languages and tools; however, the mature ones aren't simply languages to describe protocols. Mature protocol tooling generates implementations for many target languages and runtimes, and the generated code fits well into the target language and runtime. It should also be possible to use the protocol tooling to derive as much value as possible: think documentation, tests, naive implementations, and many more.

Protocol Buffers

This system uses the Google Protocol Buffers[protobuf] as the protocol language as well as the protocol implementation. The Protocol Buffers tooling generates code that not only performs the core serialisation and deserialisation functions, but includes enough metadata to allow us to treat the Protocol Buffers definitions as a domain-specific language parsed to its abstract syntax tree. Using this AST, we can easily construct mock responses and build generators for property-based testing. Turning to the lines of code in Example 1-2, we can see that the source code for simple message definition is easy to understand.

Example 1-2. Message formats

```
syntax = "proto3";
package com.reactivearchitecturecookbook.ingest.v1m0;

message IngestedImage {
  string mime_type = 1;
  bytes content = 2;
}
```

Before we move on, notice the package definition. Ignoring the root package (`com.reactivearchitecturecookbook`), we have the name of the microservice (`ingest`), followed by a version-like string formatted as `v{MAJOR}m{MINOR}`, which is rather clumsy, but necessary. We must typically start a definition of a package or

namespace with a letter, and if we used only digits after the initial `v`, we'd find it impossible to distinguish between versions `11.0` and `1.10`, for example.

A note on naming

We recommend using underscores for the field names. Taking the `IngestedImage` definition from Example 1-2, the protocol-specific public members that the C++ generator writes are `void set_mime_type(const std::string&)`, `void set_mime_type(const char*)`, `void set_mime_type(const void*, size_t)` and `void set_content(const std::string&)`, `void set_content(const char*)`, `void set_content(const void*, size_t)`. The generators for Java, Scala, and Swift turn the underscores into camel casing: the generated Scala code is based on immutable structures, giving us `case class IngestedImage(mimeType: String, content: ByteString)`. Similarly, the JSON formatter replaces the underscores by camel casing, yielding `{"mimeType":"a","contnet":"b"}` from the matching Protocol Buffer-generated instance of `IngestedImage`.

Message meta-data & envelopes

Having clear protocol definitions allows us to be very precise about the inputs and outputs of each microservice, and the structure of the messages that travel on our queues. Protocol Buffers furthermore gives us efficient representation of the messages with respect to the sizes of the serialised messages [1]. The messages such as the ones defined in Example 1-2 are sufficient for the face extract vision microservice to do its task, but it does not contain enough information for the system to correlate the messages belonging to one request. To do this, we must pack the message in an `Envelope`, defined in Example 1-3.

Example 1-3. Envelope

```
syntax = "proto3";
package com.reactivearchitecturecookbook;

import "google/protobuf/any.proto";

message Envelope {
  string correlation_id = 1;
```

1 There are more efficient protocol toolkits, but we found Protocol Buffers to have the best tooling to generate implementations in various languages, and flexible runtime to be able to serialise and deserialise the Protocol Buffers-defined types in different formats (e.g. JSON).

```
  google.protobuf.Any payload = 4;
}
```

The messages that our system processes, are the `Envelope` instances with the matching message *packed* into the `payload` field and with stable `correlation_id` throughout the system. The tooling for Protocol Buffers is available for most common languages; the tooling we care about initially is a way to generate code for the messages in the language we use. Example 1-4 shows a CMake generator, which takes the protocol definitions from the `protocol` directory

Example 1-4. CMake C++ generator

```
include(FindProtobuf)

file(GLOB_RECURSE PROTOS ${CMAKE_CURRENT_SOURCE_DIR}/../protocol/*.proto)
protobuf_generate_cpp(PROTO_SRC PROTO_HEADER ${PROTOS})
set(CMAKE_INCLUDE_CURRENT_DIR TRUE)
include_directories(${PROTOBUF_INCLUDE_DIR})
```

The tooling is similarly easy to use in Scala, to have the Scala `case classes` generated, we add the code in Example 1-5 to our `build.sbt` file.

Example 1-5. sbt Scala generator

```
scalaVersion := "2.12.1"
libraryDependencies += "com.trueaccord.scalapb"
                    %% "scalapb-json4s"
                     % "0.1.6"

libraryDependencies += "com.trueaccord.scalapb"
                    %% "scalapb-runtime"
                     % com.trueaccord.scalapb.compiler.Version.scalapbVersion
                     % "protobuf"

PB.includePaths in Compile ++= Seq(file("../protocol"))
PB.protoSources in Compile := Seq(file("../protocol"))

PB.targets in Compile := Seq(
  scalapb.gen(flatPackage = true) -> (sourceManaged in Compile).value
)
```

The source code in Example 1-4 and Example 1-4 both refer to protocol definitions in the `../protocols` directory; in other words, a directory outside of each project's root. We have taken this approach to allow us to keep *all* protocol definitions in one repository, shared amongst *all* microservices that make up the system. The directory structure of this `../protocols` directory is shown in Example 1-6.

Example 1-6. Directory structure for the protocols

```
protocol
  com
    reactivearchitecturecookbook
      envelope.proto
      ingest-v1m0.proto
      faceextract-v1m0.proto
      faceextract-v1m1.proto
      ...
      faceextract-v2m0.proto
```

Property-based testing

The generated Protocol Buffers code contains enough information about the source protocol to allow us to use them in property-based testing. Property-based testing is an approach where we describe properties that must hold for any inputs, and we use a property-based framework to take care of generating the values. To illustrate this, suppose you want to test the serialisation and deserialisation code for the `IngestedImage` message. You could do this by imagining all possible instances, but you will most likely miss out some elusive instance: one with empty `content`, one with strange `mime_type`, and so on. The essence of the test is in Example 1-7.

Example 1-7. The essence of the test

```
// Given arbitrary gen instance...
const ingest::v1m0::IngestedImage gen;

// ...we expect the following to hold.
ingest::v1m0::IngestedImage ser;
ser.ParseFromString(gen.SerializeAsString());
ASSERT(ser.content() == gen.content());
ASSERT(ser.mime_type() == gen.mime_type());
```

Using Protocol Buffers metadata, Rapidcheck[rapidcheck] and GTest[gtest], we can write a property-based test that exactly matches the essence of the test. Example 1-8 shows the *entire* C++ code.

Example 1-8. The actual test

```
#include <gtest/gtest.h>
#include <rapidcheck.h>
#include <rapidcheck/gtest.h>
#include "protobuf_gen.h"
#include <ingest-v1m0.pb.h>

using namespace com::reactivearchitecturecookbook;

RC_GTEST_PROP(main_test, handle_extract_face,
```

```
            (const ingest::v1m0::IngestedImage &gen)) {
    ingest::v1m0::IngestedImage ser;
    ser.ParseFromString(gen.SerializeAsString());

    RC_ASSERT(ser.content() == gen.content());
    RC_ASSERT(ser.mime_type() == gen.mime_type());
}
```

Example 1-9 shows a few of the generated instances (`const ingest::v1m0::IngestedImage&`) given to our test. Would you want to type hundreds of such instances by hand?

Example 1-9. Generated values

```
mime_type = , content = *jazdfwDRTERVE GFD BHDF
mime_type = .+*-<,7$%9*>:>0)+, content = \t\r\n\n\aE@TEVD BF
mime_type = )< ?3992,#//(#%/08),/<<3=#7.<-4), content = \0\13ZXVMADSEW^
...
```

The Scala tooling includes ScalaCheck[scalacheck], which works just like Rapidcheck in C++: both use the type system and the metadata in the Protocol Buffers-generated types to derive instances of *generators*, and then combining these to form generators for containers and other more complex structures. Both frameworks contain functions for further refining the generators by mapping over them, filtering the generated values, etc.

Having precise definitions of the protocols is absolutely crucial, because it allows us to precisely describe the inputs and outputs, but a good protocol tooling gives us much more. If the code that the protocol tooling generates includes sufficient amount of metadata, we can treat the metadata as an AST and use that to implement generators in property-based tests. Being able to generate valid instances of the messages also allows us to very quickly construct a *walking skeleton* of the system, where all the microservices are connected together, using the infrastructure we architected, with every code change triggering the continuous integration and continuous delivery pipeline. The only thing that is missing is the real implementation[2].

All microservices in the Image Processing System rely on explicit protocol definition, both the C++ and the Scala microservices use the Protocol Buffers toolkit to generate code to conveniently construct the values defined in the protocol files and to serialize these values into the appropriate wire format.

2 How hard can that really be?

Authentication and authorisation

Following the diagram on Figure 1-4, we need to make sure that we only accept authorized requests. More specifically, we need to make sure that *each service* is able to decide whether it is authorized to process the consumed message. The authorisation mechanism needs to give us more information that simple `Boolean`. Consider the output of our processing pipeline; all that we need to do is to make HTTP POSTs to a URL that belongs to the client (we mean someone interested in collecting the output our system produces). That sounds simple enough, but how do we compute the URL where the POSTs should be sent? We certainly need to identify the client in all messages, all the way from the `ingestion` microservice to this microservice.

The easiest approach would be to require every request hitting the `ingestion` microservice to include the URL where the responses should be pushed. While trivial, this is a terrible design choice: it allows anyone to direct the results to a URL of their choice. If this system processed sensitive information (think security camera images, identity and travel documents, or indeed performed some biometric processing), the ability to specify arbitrary URL in the request for the delivery of the results will result in leaking of such sensitive data; even if privacy did not concern you, the ability to specify arbitrary URL will result in attackers using this system to perform DOS-style attacks on the given URL.

The second approach would be to include a client identifier in each request—and message—then add a database to the push microservice, which would perform mapping from the client identifier to the URL. This would remove the DOS attack security *hole*, but would still leave us exposed to leaking data to the wrong client. This is a typical identity management, authentication, and authorisation scenario.

Let's assume we have identity management and authentication services, and explore the authorisation approaches. In monolithic applications, we typically relied on server-side session. A client would authenticate and upon success, we stored the details of the authentication in the server's memory[3] under a unique identifier. The client typically stored this identifier in a cookie, which it presented on every request. This allowed the code on the server to look up the authentication in the session; the authorisation value was used to authorise the requests. This represents a *reference-based authentication*: in order to access the authentication value, we need a client-side value (the cookie) and a service, which can resolve the reference to the authentication value. In a distributed system, we need to move to a *value-based authentication*, where the client-side value is the same as the server-side value, and can be directly used for authorisation.

3 The session was typically kept in volatile memory, but sometimes kept in a more persistent store to allow for efficient load balancing.

A good way to think about this is the difference between using a card versus using cash to pay for services. The card payment is the reference-based scenario, where cash payment is the value-based one. Glossing over some details, if someone pays by card, the merchant has to use an external system (bank) to turn the card details into usable payment. Without this external system, there is no way to find out whether the card details are convertible into the payment. With cash payment, all that the merchant has to do is to verify that the currency contains the required security elements. If so, the cash *is* the payment without further conversions.

We build a token whose payload encodes the entire authentication detail and include a mechanism to verify the token's authenticity without the need of any further services—the answer to the question "have *we* issued this *exact* token?" We require it to be passed to the `ingestion` service, and include it in every message in the system. This token can be used to authenticate the requested operations or access to the requested resources. The verification mechanism can use digital signature to verify that the token is indeed a valid token. While this allows us to verify that no-one has tampered with the token, it allows everyone to examine the token. An alternative is to use asymmetric encryption, where we encrypt the token using a public key, decrypt using a private key. A successful decryption means that a matching public key was used to encrypt it; consequently, that the token has not been tampered with. However, every microservice that needs to decrypt the token must have access to the matching private key[4].

Adding the token to our messages is a great showcase of how important it is to have well-defined protocols, and how important it is for the protocol tooling to have good support for all the languages that we use in our system. The `Envelope` with the added `token` field is shown in Example 1-10.

Example 1-10. Envelope with added token

```
syntax = "proto3";
package com.reactivearchitecturecookbook;

import "google/protobuf/any.proto";

message Envelope {
  string correlation_id = 1;
  string token = 2;
  google.protobuf.Any payload = 3;
}
```

4 Implementation of good key management system would fill the rest of this book; explore the AWS Key Management Service for inspiration.

The `ingestion` microservice extracts the value for the `token` field from the `Authorization` HTTP header (using the `Bearer` schema), and is the first service to verify that the token is indeed valid and that it contains the authorisation to access the `ingestion` service. We use the JSON Web Token defined in RFC7519 [rfc7519], but explained in a much more user-friendly way at *https://jwt.io*.

The JSON Web Token allows us to define any number of claims; think of each claim as *the bearer claims to have authorisation to do x*, where *x* is a value that the each microservice understands. In the system we're building, we use a simple naming convention for the claims: if the bearer is authorized to use the 1.*x* version of the `faceextract` microservice, the token contains the `faceextract-1.*` claim; if the bearer is authorized to use any version of the `ocr` microservice, the token contains the `ocr-*` claim. The value of these claims is specific to each microservice. Version 1.0 of the `faceextract` service does not need any further information about a claim, a simple `Boolean` is sufficient; the latest version of the OCR microservice needs complex configuration for the OCR features the bearer is authorized to use. This is a very important aspect of using token-based authorisation: the authorisation can contain very specific details and settings.

Don't create mini-monoliths

It is tempting to now construct a single *identity with configuration* management service. However, recall that the aim of a reactive microservices architecture is to decouple the services and to limit the impact of an individual service failure.

All microservices in the Image Processing System use encrypted JSON Web Tokens, which adds the complexity of good key management system (the source code includes the private and public keys as files, but that is certainly not a good practice for highly secure systems), but prevents the clients from examining the payload in the token. This system allows the end devices (think mobiles, IoT cameras, etc) to perform their processing to improve the user experience, but it *does not trust the conclusions of any classification code on the devices*; the final conclusions are computed entirely within this system. Again, the JWT libraries are available for C++ as well as Scala / Java.

Event-sourcing

Services that maintain state (particularly in-flight state), but are also able to recover from failures by restarting and reaching the same state as the failed instance, the services need to be able to re-process messages starting from the last known good message.

Depending on the communication mechanism between the microservices, we either have to provide each micoservice with its own event journal, of—if the message broker supports it—we can use the broker as the event journal[5]. Regardless of the event-sourcing mechanism (the message broker or each microservice's event journal), the flow of processing events and writing snapshots and offsets into the journal remains the same.

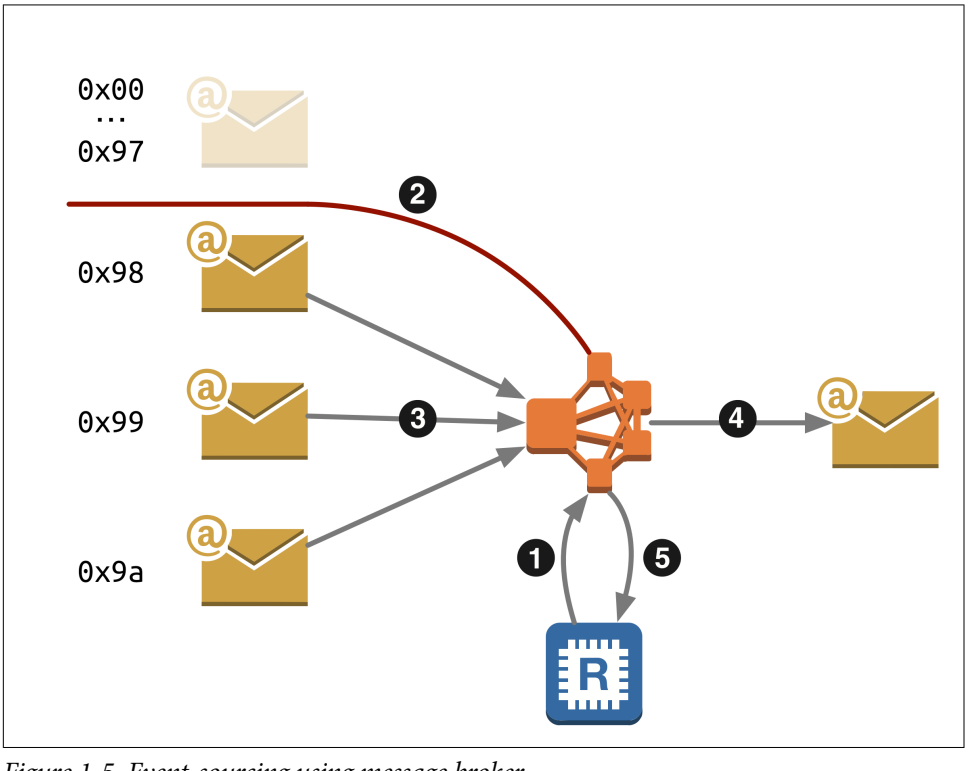

Figure 1-5. Event-sourcing using message broker

❶ Upon start, the microservice loads the offset of the last message from where it needs to consume messages in order to arrive at a well-known state. In this case, the microservice needs to consume three messages before it can produce one output message; upon producing the output message, its state is empty. (This is a special case of a *consistent* state, which can be persisted as a snapshot.)

❷ The service subscribes to receive messages from the broker starting from the loaded offset,

5 Most brokers have a time-to-live counter for the messages they keep, typically measured in units of days.

❸ The broker delivers the messages to the microservice; if the service fails during the processing of the messages, its new instance will start again from step <1>. Your runtime infrastructure should detect and prevent process thrashing, where the service keeps restarting, because the crash is triggered by one of the messages.

❹ The service has processed all three input messages, its state now allows it to produce one output message; the broker acknowledges receipt of the message.

❺ When the output message is successfully placed on the output, the microservice can write the offset `0x98` to its offsets store.

If there is a very large difference in processing complexity between consuming and validating messages and acting on the consumed messages, or if there are great variations in the velocity of the incoming messages, it will be necessary to split the microservice into the *write* and *read* sides. The write side treats the incoming messages as *commands*. The write side validates the command and turns in into an *event*, which is appended to the journal. The write side should not contain any logic responsible for dealing with the events: its responsibility is to turn the incoming command into an event in the journal as quickly as possible. The read side consumes the events the write side has appended to the journal (automatically with some delay or explicitly by asking for updates), and performs its logic. Importantly, the read side cannot append to the journal: it is responsible for acting on the events. Splitting the microservice into the two parts allows us to scale each part differently, though the exact rule for scaling depends on the exact nature of the microservice, though your aim is to balance the throughputs of the read and write sides.

Journals in message brokers

If you are using message brokers that provide reliable message delivery, allowing you to write code similar to Example 1-11, your system will still need to rely on a journal of messages and an offsets store.

Example 1-11. Implicit event-sourcing

```
val broker = Broker()
broker.subscribe("topic-1") { ❶
  case (messages, offsets) => ❷
    handle(messages) ❸

    broker.confirm(offsets) ❹
}
```

In the pseudo-code above, we define a subscription to messages on a given topic <1>; the broker delivers messages with offset that it expects us to confirm on successful processing in <2>. We handle the incoming messages in <3>, when our work is done,

we confirm the receipt of the messages in <4>. If the service fails to confirm the offsets within a timeout configured in the broker, the broker will deliver the messages to another subscriber. In order for the broker to be able to do this reliably, it cannot simply keep the messages to be delivered in memory without persisting the last confirmed offset. The broker typically uses a distributed offsets store, making most of the fact that offset is an integer (and not some other complex data structure); it can use CRDT to ensure that the cluster of offset stores eventually contains a consistent value of the offset.

Unfortunately, reliable event-sourcing and distributed offset stores do not provide a solution for situations where the messages in the journal are lost in a serious failure. Moreover, having just one journal for the entire broker (or even for individual topics) would not be sufficient for large systems.

The C++ vision libraries use implicit event sourcing by having the broker take care of re-deliveries in case of failures. The summary service, because it may have to wait for a long time for all messages to arrive to allow it to emit the response, uses the broker as the journal but maintains its own offset store. Finally, the push microservice uses its own journal and its own offset store.

Partitioning and replication

The nature of offset store means that it is a good approach to divide the broker into several topics, the values in the offsets store refer to individual topics. Even in modestly large systems, the messages in one topic would not fit in one node (*fit* refers to the I/O load and the storage space to keep the payloads, even if old messages are regularly garbage-collected), so a topic has to be *partitioned*. Each topic partition has to fit on one node (think durable storage space for the messages until they are garbage-collected); a convenient consequence is that because the messages in a topic partition are on one node, there can be deterministic order of the messages.

Partitioning helps us with distributing the messages (and the associated I/O and storage load) on multiple broker nodes; we can also have as many consumers as we have partitions, allowing us to distribute the processing load. However, partitioning doesn't help us with data resilience. With partitioning alone, we cannot recover catastrophic failures of partitions. To do so, we need to *replicate* the topic partitions: replication ensures that we have copies of the messages in the partitions on multiple nodes. The price of partitioning is loss of total order; the price of replication is that we have to think about the balance of *consistency*, *availability*, and *partition tolerance*. We would like to have all three, of course, but we can only have two *strong* properties. Fortunately, the three properties of distributed systems are not binary: there are multiple levels of consistency, which influences the degree of availability and partition tolerance. The complexity of selecting the right CAP values is a decision for the engi-

neering and business teams; it is a game of trade-offs. Nevertheless, with appropriate partitioning and replication, we can provide elasticity, resilience, and responsiveness.

Limiting impact of failures

Good protocols, value-based semantics, event-sourcing (and, where applicable, CQRS) all serve to allow the services to remain available even in failure conditions. Let's tackle failures that might seem insurmountable, but with careful technical and business consideration, we can define graceful degradation strategies.

Let's consider failure catastrophic failure in the database that contains the identities for the authentication service to use. If the business requires that users are able to log in *regardless of how degraded the system as a whole becomes*, you should consider allowing the authentiction service to issue the authentiction token for a *typical authentication details* without actually performing any authentication checks. The same applies to the authorisation service: if its dependencies fail, consider issuing very restricted *allow typical usage* token, regardless of what was passed in. The risk we are taking on here is that the system grant access to users that should not have been allowed in, but the damage to the business would be greater if legitimate users were not allowed to use the system.

The failure of outward-facing systems is easiest to describe, but there can be just as severe internal failures that our microservices can tolerate and where the impact on the business is well-understandable. The event-sourced microservice can tolerate failures of its offsets store. The business decision will drive the length of time it can tolerate the failure for, and what it does if the failure is persistent or permanent. If the microservice is running, and the offset store becomes unavailable, we risk having to re-process growing number of messages in case of failure or scaling events.

The business impact is defined by the computational cost to re-process already successfully processed messages (remember, we could not write the latest offset to the offset store!), and the impact on the customers who will receive duplicate messages. Similarly, if the offset store is unavailable during the microservice's startup, the business decision might be to start processing at offset defined as offset_last-100, or even zero. The business risk is that some messages might not be processed, or that there will be too many needless messages re-processed. Nevertheless, both might be perfectly acceptable compared to the service being unavilable.

Good example of limiting the impact of failures is the summary microservice, which tolerates temporary failures in its offset store and the push microservice, which tolerates temporary failures in its journal. The authorisation microservice tolerates total failures of its dependencies: in that case, the service issues *allow every idempotent workload* tokens to the clients, which the business deemed to be a good graceful degradation strategy. The clients can still submit images to be processed—these idempotent requests—but non-idempotent requests are not authorized. The tokens have

short expiry date with random time delta added to each one. The clients to refresh the tokens on expiry, but the random time deltas in the expiry dates spread the load on the authorisation service once it recovers.

Back-pressure

Without back-pressure, the information flows in the system only in one direction, from source to sink; *the source sets the pace and the sinks just have to cope*. If the system is balanced, then the sinks can cope with the data the source produces, but a spike in usage will disturb this balance. Let's explore what happens when the upstream service accepts requests from external systems and is capable of handling much greater load than the downstream service in Figure 1-6.

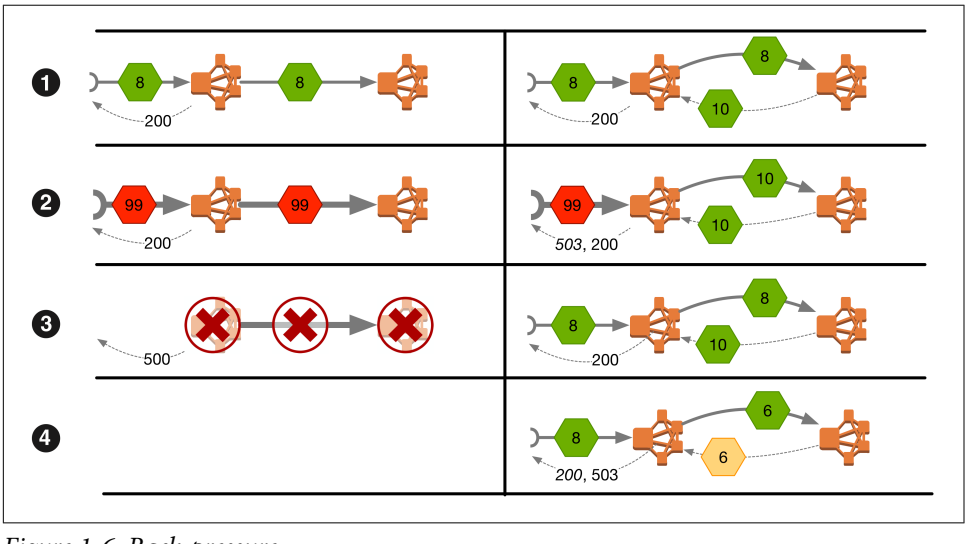

Figure 1-6. Back-pressure

❶ In happy-days scenario, the upstream service receives requests from external systems at the rate that the downstream service can process. Without back-pressure, the system *happens to work*, because the load is within the predicted range. With back-pressure, the system also works well, because the downstream service tells the upstream component how much load it is ready to consume.

❷ When there is a spike in the incoming load, the situation changes dramatically. Without back-pressure, the upstream accepts all requests, then overwhelms the downstream service. With back-pressure, the upstream service knows that *downstream services can only process 10 messages*, so it rejects most of the incoming requests, even though *it* could handle all the extra load. Replying with errors is

not great, but it is better than causing failures that spread through the entire system.

❸ The cascade of failures spreads from the downstream service, which can no longer handle the load pushed to it. This might cause failures in the message broker (its partitions grow too large for the nodes that host them), which ultimately causes failures in the upstream component. The result is that the system becomes *unavailable* (or *unresponsive*) for everyone. In case of REST API, this is the difference between receiving HTTP 503 (over capacity) immediately, or waiting for a long time and receiving HTTP 500 (internal server error) or no response at all.

❹ Systems with back-pressure can flex their capacity if there are intermittent capacity problems or transient failures; the upstream services remain available and responsive, even if that means rejecting requests that would take the system over its immediate capacity.

It isn't possible to dismiss back-pressure as irrelevant simply because the system uses a message broker to connect two services. While it is true that the broker can provide short-term buffer for systems without back-pressure; nevertheless, if the downstream messages do not keep up with the incoming message rate in the broker, the time for a specific message to travel through the system will increase. If the increase in load is not just a momentary spike, the downstream services will be busy dealing with queued messages and starting to breach the performance service levels, while newer messages are being queued up; their performance service level is breached even before they reach the downstream service. Put bluntly, you will have a system whose services are running at full capacity, without failures, but the result is nothing but timeouts. Before we conculde, it is important to point out that, even though the back-pressure portion of Figure 1-6 does not explicitly show it, the back-pressure reporting *must be asynchronous*. If the upstream service blocks on the *poll* operation to find out how much the downstream component can accept, then the upstream service becomes unresponsive.

All microservices in the Image Processing System use back-pressure in the libraries that connect them to the message broker; the `ingestion` and push microservices rely on Akka HTTP to manage back-pressure using the underlying asynchrnous I/O that handles the incoming and outgoing network connections.

External interfaces

Within our own system, we can implement back-pressure from start to end, but we have to take into account external systems. We already know that we don't control the load we receive from the external systems, so it is important to reject requests that would result in overloading our system. Even though our system is in the position of

some external system for our integration clients, we should do our best to not cause a denial-of-service "attack" on our clients.

Because we can't assume any details about the client systems, we have to start from the position of no back-pressure reporting from the client. We have to take cues from the underlying I/O to deduce the acceptable load[6]. When we detect that we are overloading the client (because we are beginning to get timeouts or error status codes), it is important not to make things worse by aggressively re-trying the outgoing requests. This means that we have to consider some type of dead-letter store, where we keep the failed requests to retry later. Moreover, a push-style interface requires the clients to have a public-facing endpoint. This endpoint should require secure transport, which is trivial to implement, but it also means that our system is responsible for checking that the connection is indeed secure. And so we have to verify the endpoint's certificate, which means that we are now responsible for maintaining the certificate chains, checking revocations, etc; and it takes only one client to request that we handle self-signed certificate or clear-text connections to significantly increase the complexity of the push service.

Firehose

An alternative approach is to have a firehose API, where the clients open a long-running HTTP POSTs to our systems, indicating the last-successful offset (and other parameters such as maximum data rate and any filters to be applied). Our system then sends response chunks that represent batches of messages (the same ones that the push microservice would send), which means that our clients don't have to do as much work to implement public-facing endpoint, and they are ultimately in control of the data rate: the client's ingestion system can drop the connection whenever it wants.

In both cases, the push and firehose services need to maintain their own journals. Relying on the broker's message journal would work well if we had well-known and fixed number of clients (allowing us to structure the queues / partitions appropriately), but we hope that the number of clients will grow. This means that we should maintain our own journal, disconnected from the broker's journal, which means message duplication (a message is stored in the broker's journal as well as the microservice's journal; moreover, the services cannot share their data stores, which means that the same message is in the firehose and push journals). However, we gain finer control over partitioning and message grouping. Figure 1-7 shows the final architecture.

6 It is possible to use predictive models that tell us whether the load we are sending to a particular client is growing past safe load.

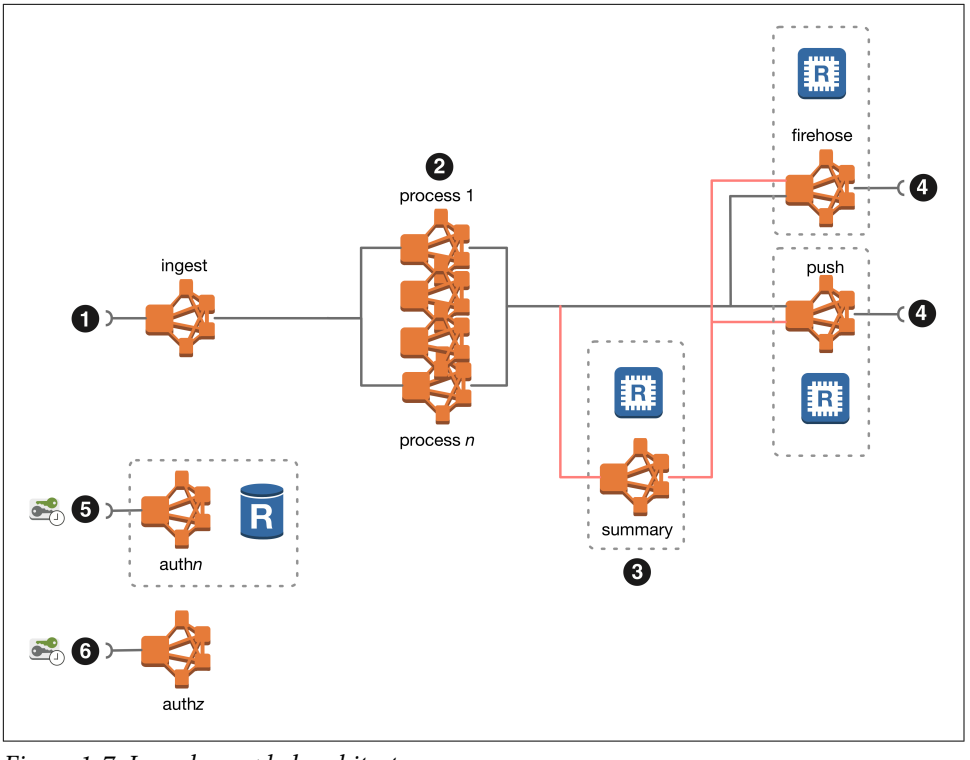

Figure 1-7. Loosely coupled architecture

We are now ready to explore the important implementation details in each of the services.

Implementation

Now that we have explored the main architectural concepts, it is just as important to explore the implementation. We will no doubt have to make compromises in the implementation, but it is important to understand the impact of the compromises. As we work on the project, we need to periodically review the decisions that led to the compromises we made to make sure that the impact of the compromises, which we presented to the business, remains the same. With that in mind, let's see how to implement the system we architected.

All services except the computer vision services are implemented in Akka and Scala. Akka brings us low-level actor concurrency, but also includes convenient high-level programming models that handle HTTP REST servers and clients and stream processing. The computer vision services comprise the actual vision code that relies on OpenCV and CUDA. We considered implementing JNI interfaces for the native code, allowing us to implement the "connectors" in Akka and Scala, but after comparing

the complexity of the JNI work with the complexity of using (modern) C++ client libraries for our message broker, service discovery, and others, we decided to implement the vision services without any wrappers.

We used Apache Kafka as our message broker, though we briefly considered AWS Kinesis[kinesis]. Apache Kafka has the capability to handle a large number of messages, and it can be used as (short-term) event journal for our microservices. One slight downside is that Apache Kafka requires you to define the partitioning up-front; it does not automatically vary the number of partitions depending on the number of available nodes. It is possible to add or remove partitions, but you have to handle draining of the partitions you are removing, which may be difficult if you are using automatic partition assignment derived from message key hashes—you will have to ensure that you do not publish messages with keys whose hashes would direct them to the partitions you are draining. (For large-scale deployments, you may wish to compute the partition number in your code.) Increasing the number of partitions is easier, though only new records will be partitioned; Kafka does not re-partition the existing topics. Other brokers follow different naming, but similar semantics.

Ingestion microservice

The `ingestion` microservice is responsible for accepting the requests containing the images from the clients, checking that the client is authorized to make the request, then checking that the requests contain valid data (i.e. is the HTTP POST body really an image?); finally, the `ingestion` microservice packs the validated payload into our envelope and publishes it to an outgoing queue. Once all of this work is done, the microservice replies to the caller with a simple acknowledgement response. If the client receives HTTP 200, we have validated and ingested the request, and we will complete its processing at some point in the future. As far as the client (think another system, web, or mobile application using our REST API) is concerned, it will not receive any further communication.

Example 1-12. Ingestion route

```
trait IngestionRoute extends Directives with GenericUnmarshallers { ❶
  def authorizeAndExtract(token: String): Try[JWTClaimsSet] ❷
  def extractIngestImage(request: HttpRequest)
                        (implicit ec: ExecutionContext): Future[IngestedImage] ❸
  def publishIngestedImage(claimsSet: JWTClaimsSet, token: String,
                           transactionId: String)
                          (ingestedImage: IngestedImage)
                          (implicit ec: ExecutionContext): Future[Unit] ❹

  private def authorizedRouteHandler(requestContext: RequestContext,
                                     token: String, transactionId: String)
                                    (claimsSet: JWTClaimsSet)
                                    (implicit ec: ExecutionContext): Future[RouteResult] = { ❺
```

```
    extractIngestImage(requestContext.request)
      .flatMap(publishIngestedImage(claimsSet, token, transactionId))
      .flatMap { _ => requestContext.complete("") }
      .recoverWith { case x => requestContext.fail(x) }
  }

  def ingestionRoute(implicit ec: ExecutionContext): Route = ❻
    path(RemainingPath) { tid => ❼
      post { ❽
        extractCredentials { ❾
          case Some(credentials) if credentials.scheme() == "Bearer" =>
            request: RequestContext => ❿
            authorizeAndExtract(credentials.token())
              .map(authorizedRouteHandler(request, credentials.token(), tid.toString()))
              .getOrElse(request.reject(AuthorizationFailedRejection))
          case _ => _.reject(AuthorizationFailedRejection)
        }
      }
    }

}
```

❶ We mixin the `IngestionRoute` trait to the *main* class and implement the steps in the ingestion pipeline in the abstract methods; notably, they all return `Futures`, implying non-blocking implementations.

❷ The `authorizeAndExtract` function takes the `String` that claims to be the JSON Web Token the authorization service issued. The implementation is expected to parse it, validate that it was indeed the authorization service that issued it, and return the `JWTClaimsSet` wrapped in `Try`.

❸ The `extractIngestImage` function takes the `HttpRequest` and is responsible for checking that the request body does indeed contain bytes that are a valid image. It returns a `Future[IngestedImage]`, allowing the implementations to perform complex asynchronous validation, if needed.

❹ The `publishIngestedImage` is responsible for taking—amongst others—the `IngestedImage` and placing it on the output topic. It returns the `Future[Unit]` once it receives confirmation from Kafka that the message has been placed on the requested topic. (We say `Future[Unit]` because we aren't interested in the value of the result; we only want to be able to tell the difference between `Success(())` and `Failure(())`.)

❺ The `authorizedRouteHandler` implements the processing pipeline that is applied to the incoming HTTP requests.

❻ A `Route` is a *function* mapping `RequestContext` to `Future[RouteResult]`; we construct this function by using Akka HTTP's DSL.

❼ The `path` directive matches the request path on its parameter; in this case, we want the entire remaining path.

❽ We only accept the HTTP POST method, so we use he `post` directive.

❾ To extract the credentials supplied with the request (in the `Authorization` HTTP header), we apply the `extractCredentials` directive and then match on its optional result.

❿ We finally use the steps in our processing pipeline by checking the authorization token first, then mapping the successful value through the `authorizedRouteHandler`; failures result in the appropriate HTTP rejection

It is important to notice the asynchrnous nature of the entire processing pipeline. The `ingestionRoute` returns a function that Akka HTTP uses in its low-level actor runtime to service the incoming HTTP requests. Akka HTTP handles not just the incoming route, but also bakes in back-pressure management and reporting.

Vision microservices

The vision microservices each perform different computer vision task on the same input data. Because the computer vision code is implemented in C++ & CUDA, and because we want to keep the complexity of the microservice to a minimum, we are not going to create wrappers that make the native code accessible from Java or Scala code. After all, one of the points of microservices is that we should be able to use the most appropriate language or toolkit for each microservice.

Regardless of the language, the microservice fit into our message-driven infrastructure. The computer vision work results in zero or more output messages that go on the output Kafka topic. Before we consider the implication of this design, let's review the core principles of Kafka messaging: we need to understand topics, partitions, and consumer groups.

A *topic* is a concept of a basic pipe, identified by some name, that transports the bytes that make up the payloads of the messages from producers to consumers. Kafka parallelises the message delivery by partitioning the topic.

A *partition* is essentially a directory with journal files for the messages; a partition has to fit on one node in terms of the size of the journal and the I/O performance. (If you have one partition on a particular topic, that topic will not scale on multiple nodes, even if they are available; if you have a topic with 1000 partitions, you could potentially make use of the processing and I/O power of 1000 nodes.) Because one parti-

tion is fully contained in one node, each partition is totally ordered; however, a topic with more than one partition is not totally ordered. The partition count therefore defines the consumer parallelism: we can consume as many batches of messages as there are consumer groups. We can grow the number of partitions, but the existing messages in the topic will not be repartitioned.

The consumption of messages is handled by a *consumer group*, which is a set of threads that receive the messages in the partitions. Kafka selects exactly one thread to deliver it a batch of messages from a partition.

To help us think about the messaging in Kafka, and to help us understand topics, partitions and consumer groups, we say that:

- topics are divided into partitions; the number of partitions defines the maximum consuming parallelism,
- each consumer group receives the same partitions (containing the same messages),
- if there are more consumer threads than there are partitions on a topic, some threads will never receive a message,
- if there are more partitions than threads, some threads will receive data from multiple partitions,
- if a consumer thread subscribes to multiple partitions there is no guarantee about the order the received messages, other than that within the partition the offsets will be sequential. For example, the thread may receive 5 messages from partition 10 and 6 from partition 11, then 5 more from partition 10 followed by 5 more from partition 10 even if partition 11 has data available,
- adding or removing the consumer group threads will cause Kafka to re-balance, possibly changing the assignment of a partition to a thread,
- in high-level consumers, Zookeeper maintains the offsets of the delivered messages for each partition in a consumer group,
- in low-level consumers, the consumers are responsible for maintaining the offset stores, allowing them to control the offset from which they want to subscribe.

Figure 1-8 illustrates three consumers subscribed to one partition within a consumer group.

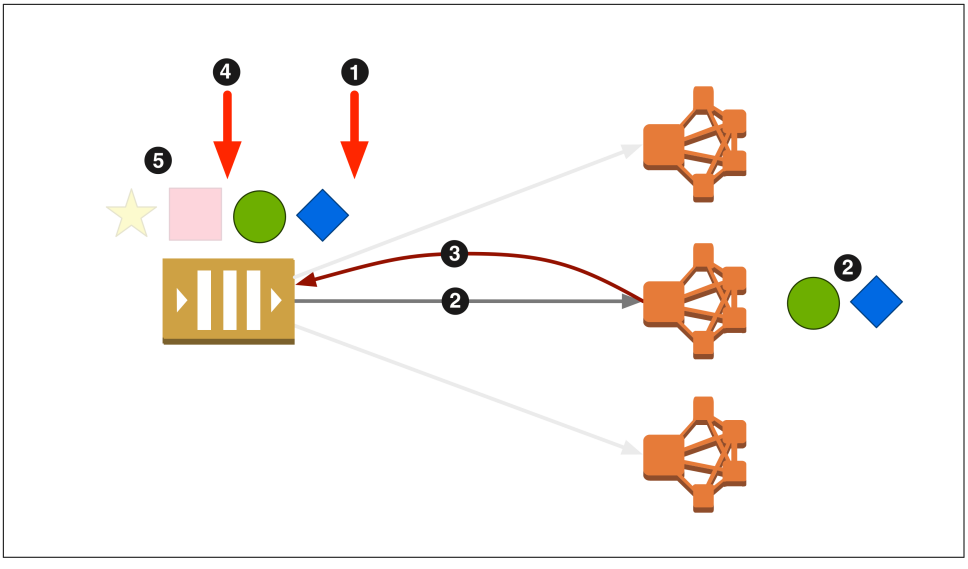

Figure 1-8. Kafka messaging

Let's explore how the messages are consumed:

❶ We are starting when two new messages in a topic partition arrived after the last offset,

❷ Kafka selects one consumer thread (from a known set of consumer threads it obtained when re-balancing the partitions) and delivers it a batch of messages, the consumer processes the batch of messages and,

❸ confirms the processed offsets within the configured timeout (which should be as small as possible),

❹ the new offset is moved, effectively marking the messages in the journal as processed;

❺ when new messages arrive, the process repeats.

Confirming the consumption not at the point of message arrival, but *after* successful processing explains how we can guarantee at-least-once delivery semantics. In the happy-days scenario, we complete all our processing within the time Kafka gives us to confirm the offsets of the received batch of messages. If the batch of messages is lost on the way from Kafka to the service, or if the service receives the messages, but Kafka does not receive the confirmation within a configured timeout (because the service failed or because the confirmation was lost), Kafka will repartition the topic

and select a different consumer thread to handle the batch starting from the last known confirmed offset.

Receiving messages in code

Let's turn to pseudo code in Example 1-13 and explore how we might go about implementing Kafka consumer and producer: a consumer consumes a message from the subscribed topics from Kafka, the microservice applies its processing code and uses the producer to publish a message on the outgoing topic. When the message is published, the microservice commits the journal offset.

Example 1-13. Receiving and sending messages

```
Consumer consumer;
Producer producer;
consumer.subscribe(std::vector<std::string>{"in_topic"});
while (true) {
    auto in_message = consumer.consume();

    auto out_message = process(in_message);
    producer.produce("out_topic", out_message);
    consumer.commit(in_message);
}
```

Unfortunately, such code—while easy to understand—would not be efficient, because it is entirely synchronous and blocks on the I/O operations. Blocking on I/O operations is particularly dangerous practice, because there is no telling how long the thread will be blocked. We could solve some of this by adding timeouts and more complex control flow to the blocking functions; we would have some control over the responsiveness of the microservice, but the microservice would not be able to make the most of the computational resources of the node it runs on. We might be tempted to solve this by wrapping each of the I/O operations in its own thread. Even if we ignore the complexities of synchronisation of different flows of execution, threads are very heavy objects.

Apart from the context structures that the operating system's kernel needs to maintain for the thread, each thread needs to have its own stack, typically 1 MiB big. Thirty-two bit processes that created too many threads ran out of 32bit addresses where the thread stacks could be placed. (Recall that 32bit pointers give us 4 GiB of addressable memory; in Linux, the top 1 GiB is typically reserved for kernel addresses, leaving only 3 GiB for program addresses with the stack growing from one end of the 3 GiB block and heap growing from the other end. Even if we had no heap, we would only have space for 3000 threads!) Sixty-four bit pointers give us more addressable space, but we are still restricted by the available virtual memory space, and we typically restrict the amount of memory a process can consume, especially when running in containers.

Use non-blocking I/O

Rather than writing entirely blocking code, or using threads to contain the blocking I/O, we should use the asynchronous I/O support available in modern OS kernels and exposed through the userspace libraries. Asynchronous I/O operations complete immediately, but their result is not the result of the requested I/O operation, but a result of validating the inputs and enqueuing the operation. One of the inputs is a *callback* which the kernel will invoke when the I/O operation completes. (You control the way in which this callback executes, for example, in POSIX asynchronous I/O functions, you have a choice of receiving a signal or having the kernel run the provided callback in a separate thread.) Let's leave the pseudo-code and implement a non-blocking version of the pseudo-code in modern C++ with librdkafka[librdkafka] in Example 1-14.

Example 1-14. Receiving and processing messages

```
using commit_opaque = std::tuple<
    std::shared_ptr<std::atomic_long>,
    std::shared_ptr<RdKafka::TopicPartition>>;

class commit_dr_cb : public RdKafka::DeliveryReportCb { ❶
private:
    std::shared_ptr<RdKafka::KafkaConsumer> consumer_; ❷
public:
    commit_dr_cb(std::shared_ptr<RdKafka::KafkaConsumer> consumer) :
      consumer_(consumer) {
    };

    void dr_cb(RdKafka::Message &message) override {
      commit_opaque *co = static_cast<commit_opaque *>(message.msg_opaque());
      auto ctr = std::get<0>(*co);
      if (std::atomic_fetch_sub(ctr.get(), 1L) == 1L) {
          auto tp = std::get<1>(*co);
```

```
            if (message.err() == RdKafka::ERR_NO_ERROR)
                consumer_->commitAsync(std::vector<RdKafka::TopicPartition *>{tp.get()}); ❸
            delete co; ⓫
        }
    }
};
std::atomic_bool run(true);
const std::string in_topic_name = "ingest-1";
const std::string out_topic_name = "vision-1";
const std::string key = "key";
auto conf = std::unique_ptr<RdKafka::Conf>(RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL));
auto consumer = std::shared_ptr(RdKafka::KafkaConsumer::create(conf, ...));
commit_dr_cb dr_cb(consumer);
conf->set("dr_cb", &dr_cb, err_str); ❹
auto producer = std::unique_ptr(RdKafka::Producer::create(conf, ...));
auto out_topic = std::unique_ptr(RdKafka::Topic::create(producer.get(), out_topic_name, ...));
consumer->subscribe(std::vector<std::string>({ in_topic_name })); ❺
while (run) {
    auto in_message = std::unique_ptr<RdKafka::Message>(consumer->consume(10)); ❻
    if (in_message->err() != RdKafka::ERR_NO_ERROR) continue;
    std::string out_payload = ...; ❼
    commit_opaque* opaque = new commit_opaque(
        std::make_shared<std::atomic_long>(1L),
        std::shared_ptr<RdKafka::TopicPartition>(
          RdKafka::TopicPartition::create(
            message->topic_name(), message->partition(), message->offset()
          )
        )
    ); ❽
    const auto resp = producer->produce(out_topic.get(), partition,
        RdKafka::Producer::RK_MSG_COPY,
        const_cast<char *>(out_payload.c_str()), out_payload.size(),
        &key, opaque); ❾
    producer->poll(0); ❿
    if (resp != RdKafka::ERR_NO_ERROR) delete opaque; ⓫
    ⓬
}
consumer->close(); ⓭
producer->flush(1000); ⓮
```

❶ The *delivery report callback*, `commit_dr_cb`, class is responsible for committing the offsets on successful delivery report,

❷ it maintains a shared reference to the `RdKafka::KafkaConsumer` instance,

❸ the `consumer_` refrence is used to *enqueue the commit operation* on successful delivery report of the last message (to find out which is the last message, we are using the values in the pointer to `commit_opaque` tuple; see step 8),

❹ the instance of the `commit_dr_cb` is set in the `RdKafka::Producer` configuration,

❺ the created `RdKafka::KafkaConsumer` subscribes to the given topic names,

❻ in the processing loop, the `RdKafka::KafkaConsumer::consume` method is called on every iteration; even though it looks as though it performs the underlying "read" I/O operation, it actually *enqueues a poll* operation, and returns the first message from the polling queue or a "failed message" (where `RdKafka::Message::err() != RdKafka::ERR_NO_ERROR`) when the specified timeout elapses and there are no messages in the queue,

❼ we use the message to perform our the microservice's core work—given the nature of the vision microservices, this is CPU and GPU-bound work,

❽ we construct a pointer to the `commit_opaque` instance, which we pass to the `RdKafka::Producer::produce` method to ultimately reach the `commit_rd_cb`, where it will be used to determine which message in the response is the last one,

❾ once we have the output message, we use the `RdKafka::Producer::produce` to *enqueue a send* operation; the call returns immediately,

❿ it is our responsibility to periodically call the `RdKafka::Producer::poll` operation to begin the underlying I/O operations on the enqueued messages,

⓫ the current librdkafka only has a thin C++ wrapper, which does not rely on modern C++, so we have to use raw pointers and remember to `delete` them when appropriate to avoid leaking memory,

⓬ when the producer receives confirmation from Kafka that the message has indeed been produced, it calls the registered *delivery report callback*, where we *enqueue the commit* operation,

⓭ before the microservice exits gracefully, it must call `RdKafka::KafkaConsumer::close` to give the broker the opportunity to gracefully repartition the topic,

⓮ finally, the microservice should attempt to clear all producer queues before finally exiting.

Implementing the image processing

The code in Example 1-14 is entirely asynchronous and non-blocking, but it remains fairly understandable; especially if we make the most of the standard library in modern C++, which gives us easy-to-use, but predictable and very efficient memory and resource management primitives. Now that we have the core structure in place,

we can tidy up and expand the handling of the messages: the messages arrive as Protocol Buffer serialised `Envelope` instances; our vision microservice may produce zero or more messages as a result of processing one input message. This leads us to the code in Example 1-15.

Example 1-15. Handling real messages

```
namespace fe = faceextract::v1m0;
namespace in = ingest::v1m0;

template<typename T, typename U, typename Handler>
std::vector<Envelope> handle_sync(
    const std::unique_ptr<RdKafka::Message> &message, Handler handler) {
    ...
}
const auto partition = RdKafka::Topic::PARTITION_UA;
auto message = std::unique_ptr<RdKafka::Message>(consumer->consume(10));
if (message->err() != RdKafka::ERR_NO_ERROR) continue;
const auto key = message->key();
const auto out_envelopes = handle_sync<in::IngestedImage, fe::ExtractedFace>(
    message,
    [](const auto &, const auto &) { return std::vector<fe::ExtractedFace>(...) });
    commit_opaque* opaque = new commit_opaque(
        std::make_shared<std::atomic_long>(out_envelopes.size()),
        std::shared_ptr<RdKafka::TopicPartition>(RdKafka::TopicPartition::create(
            message->topic_name(), message->partition(), message->offset()
        )
    )
);
for (const auto &out_envelope : out_envelopes) {
    const auto out_payload = out_envelope.SerializeAsString();
    const auto resp =
        producer->produce(out_topic.get(), partition, RdKafka::Producer::RK_MSG_COPY,
                          const_cast<char *>(out_payload.c_str()), out_payload.size(),
                          &key, opaque);
}

producer->poll(0);
```

The code we have shown in Example 1-14 and Example 1-15 is the barest minimum you will need in your implementation: it is easy to understand, yet efficient. It also maintains the at-least-once delivery semantics, because we only enqueue the commit offsets operation once we have a confirmation of a successful delivery. If no messages are lost, we have exactly-once delivery. If we fail to produce the message, if the delivery report message is lost, or if the commit message is lost, Kafka will deliver the messages (starting from the last committed offset) to another thread in the consumer group. Depending on where the failure occurred, we will have exactly-once or at-least-once delivery semantics. Luckily, the vision microservices are idempotent: they

produce the same response for the same input message, regardless of how many times the input message arrives, so having at-least-once delivery semantics does not cause any problems apart from the cost of running the computation. However, there is a long way from having the code to having a well-tested microservice running in a production environment.

We attempted to implement the computer vision microservices as unikernels using IncludeOS[includeos], which, at the time of writing, encouragingly only supported non-blocking I/O. Unfortunately, we quickly realised that the unikernel does not play nicely with the Nvidia GPUs, which stopped us in our tracks.

We leave out the description of the actual work that the computer vision services perform; however, the vision functions are deterministic and *pure*, in the sense that the service processes one image at a time and that for the same image, it produces the same result. Nevertheless, the vision services are heavily CPU and GPU-bound, making the asynchronous I/O crucial in order to make the most of the computational resources of the node the service runs on. If your system calls for a vision service that uses a stateful model (think LTSM for automatic image captioning with context) that nevertheless does not ever lose messages, you will need to follow the event-sourcing approach outlined in the `summary` service, but add message de-duplication.

Push microservices

We construct the token using the standard fields, but add the `push` string claim, which holds the URL that the push microservice should use. For simplicity, we say that if the token does not include the `push` claim, then the caller is not authorized to use the push microservice. It seems that all there is left for us to do is to subscribe to the appropriate Kafka topics, consume the messages, and make the appropriate push requests in Example 1-16.

Example 1-16. Initial push implementation

```
class PushActor(jwtDecrypter: JWEDecrypter) extends Actor {
  implicit val materializer = ActorMaterializer()
  private val pool = Http(context.system).superPool[Unit]()
  private[this] val kafkaConsumerActor = context.actorOf(...)

  private def push(record: ConsumerRecord[String, Envelope]): Unit = {
    (for {
      jwt <- Try(EncryptedJWT.parse(record.value().token)) ❹
      _ <- Try(jwt.decrypt(jwtDecrypter)) ❷
      uri <- Try(Uri(jwt.getJWTClaimsSet.getStringClaim("push-1"))) ❺
      entity = HttpEntity(ContentTypes.`application/json`,
                          JsonFormat.toJsonString(record.value().payload)) ❻
```

```
      rq = HttpRequest(method = HttpMethods.POST, uri = uri, entity = entity) ❼
    } yield (rq, ())).foreach(x => Source.single(x).via(pool).runWith(...)) ❽
  }

  override def receive: Receive = {
    case MessageBatch(consumerRecords) =>
      consumerRecords.recordsList.foreach(push) ❶
      kafkaConsumerActor ! Confirm(consumerRecords.offsets)
    case (Success(resp@HttpResponse(_, _, entity, _)), _) => ❷
      resp.discardEntityBytes()
    case (Failure(_), _) => ❸
      // bad request
  }

}
```

❶ When a batch of messages arrives from Kafka, we apply the `push` function to each message; this is an asynchronous operation, so we are able to confirm the delivered offsets immediately,

❷ we must handle HTTP responses from the client, at the bare minimum, we must ensure that we consume the bytes that the client sent to allow the asynchronous I/O to complete; we must do this even if we do not care about the content of the response,

❸ we handle the failure in a no-op operation,

❹ the processing of each `ConsumerRecord[String, Envelope]` starts by checking whether the supplied token is valid,

❺ given a valid token, we verify that the `push-1` claim is present and that it can be parsed into a `Uri`,

❻ the entity that will be sent out is a JSON representation of the message's payload,

❼ the request in our simple implementation is a HTTP POST with the given entity,

❽ finally, we use Akka's pooled, cached, and back-pressure aware HTTP client to begin handling the request

This code is similar to Example 1-20, but it is clearly not handling catastrophic failures well. The underlying Akka HTTP machinery handles back-pressure from the client, with so that if we flood it with requests it cannot handle, we are notified in the actor's behaviour, but there is nothing we actually do about the errors. Moreover, if the node hosting this microservice crashes after confirming the offsets to Kafka, but

before it can receive responses (successful or not) from the client, we never go back to the messages we failed to send. (We have already confirmed the offsets, and we're in auto-partitioning mode.) "Not a problem," you say, "we'll just use the auto-partitioning mode with manual offsets, and we'll persist the offsets." This would work well if we had the as many (or more) partitions as clients; recall that we use the client identity as the partitioning key. If we have fewer partitions than clients, the *latest sent offsets* will include messages for multiple clients. While the clients no doubt understand that there could be duplicate messages, we should limit the duplicates to an absolute minimum.

Resilient push service

To implement a push service that is "nice" to our customers (it backs off in case the client's endpoint starts failing; it tries to send the failed requests later), but it does not lose any requests before it delivers them to the client, we could use the broker-backed event-sourcing approach. We would create a topic with as many partitions as we have clients and then fall back on the auto-partitioning with manual offsets mode of operation. However, this approach will not scale well if the number of clients is difficult to predict[7]. So, we need to use our own journal.

The `PushActor` becomes `PersistentActor`, which gives it the ability to write events to a journal. (The type of journal and details of how to connect to it are configurable.) Having persistence changes the flow of processing to one where as soon as we have validated the messages, persisted them as *events* in our journal, we confirm the consumed offsets to Kafka. We perform the actual HTTP push operations at some later point. We have in effect divided the microservice into *write* and *read* sides: we have a CQRS/ES microservice! (Remember that the only motivation for this complexity is because we want to implement service that can recover its state in case of failures, and because the velocity of the messages arriving might be completely different than the velocity with which we are able to output the messages.)

7 Imagine that the system is actually successful and new clients sign up in drones!

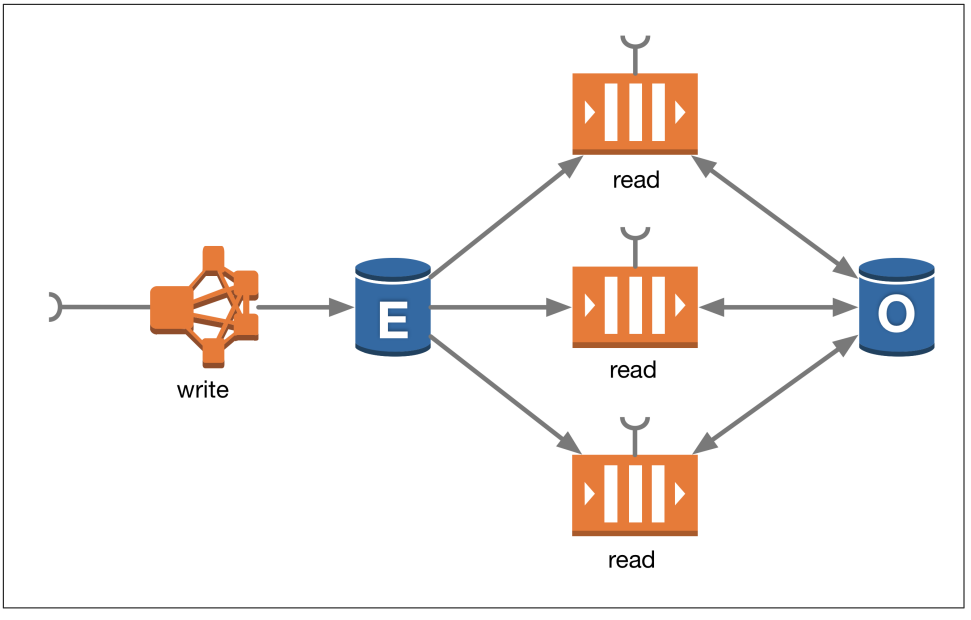

Figure 1-9. CQRS/ES

Even though CQRS/ES implementation shown in Figure 1-9 is a complex piece of code, Akka handles most of this complexity.

The write side

The write side is implemented in essence by switching from `Actor` to `PersistentActor` in Example 1-17.

Example 1-17. CQRS/ES write side

```
class PushActor(...) extends PersistentActor {
  ...
  override def receiveCommand: Receive = {
    case PushActor.extractor(consumerRecords) =>
      val requests = consumerRecords.recordsList.flatMap { ... } ❶

      persistAll(requests) { _ => ❷
        kafkaConsumerActor ! Confirm(consumerRecords.offsets) ❸
      }
  }
}
```

We are on the write side of CQRS/ES, so we treat the incoming Kafka consumer records as commands, which need to be validated using our token-based authorization, giving us the sequence of `Envelopes` in step <1>. We persist the envelopes in the

configured journal using the `persistAll` call <2>. Keep in mind that Akka is non-blocking throughout; even though the name of the `persistAll` might look blocking, it completes immediately, but the function given to it as the second argument is called when the journal write operation completes. So, <3> is the earliest we can confirm the offsets to Kafka, which completes the tasks of the write side.

The read side

The read side treats the journal as a source of events, which it can subscribe to. Without going into deep details of Akka Persistence, we will just say that we have multiple readers, each reading a different set of events from the journal, and each maintaining their own offsets store. For high-level outline of the code, see Example 1-18.

Example 1-18. CQRS/ES write side

```
object PushView {

  def apply(tag: String, redisClientPool: redisClientPool)
           (implicit system: ActorSystem, materializer: ActorMaterializer) = ... ❶

}

class PushView(tag: String, offset: Offset, redisClientPool: redisClientPool)
              (implicit system: ActorSystem, materializer: ActorMaterializer) {
  private val pool = Http(system).superPool[Offset]() ❷
  private val readJournal: RedisReadJournal =
    PersistenceQuery(system).readJournalFor[RedisReadJournal](RedisReadJournal.Identifier) ❸
  private val source: Source[EventEnvelope2, NotUsed] = readJournal.eventsByTag(tag, offset) ❹
  private def eventToRequestPair(event: Any): (HttpRequest, Offset) = ... ❺
  private def commitLatestOffset(result: Seq[(Try[HttpResponse], Offset)]): Future[Unit] = ... ❻

  source
    .map(e => eventToRequestPair(e.event))
    .via(pool)
    .groupedWithin(10, 60.seconds)
    .mapAsync(1)(commitLatestOffset)
    .runWith(Sink.ignore) ❼

}
```

❶ As convenience, we provide the `apply` function in the companion object which loads the latest offsets for the given tag from our offsets store.

❷ The instances of the `PushView` construct a super pool of requests (including by-host caching), with `Offset` as the tag of responses.

❸ We construct the `RedisReadJournal`, which will—amongst others—give us the stream of filtered events.

❹ The source of events is a subscription to the journal with restriction that compares the given tag value.

❺ The events in the journal need to be cast to the value that was read (the type of `EventEnvelope2.event` is `Any`).

❻ We receive a sequence of results of the HTTP requests, and our task is to write to our journal store the offset before which all requests have succeeded.

❼ This is the actual processing flow, which defines the transformation pipeline from source to sink; notice that we group the responses by 10 or 60 seconds (which defines the maximum number or time span of duplicate messages our system sends).

Constructing instances of the `PushView` allows us to construct different workers that actually handle the HTTP push work, but *we* control their creation at the service's startup and *we* control the offset store. This gives us the message delivery guarantees, and allows us to limit the duplicate message delivery in event of failures.

Summary service

Let's make integration easier and bring additional value to our clients by implementing a message summary microservice. This microservice can combine the output of the vision microservices—and using our knowledge of the vision processes—produce useful high-level summaries of multiple ingested messages.

The summary microservice combines multiple messages to produce one, but the input messages arrive at arbitrary points in time (potentially seconds apart) and in arbitrary order: the summary microservice has to maintain state that allows it to recover this state in case of failure. Because the summary microservice need to maintain their state over a number of seconds, we can no longer simply leave the offsets of the incoming messages *unconfirmed* in Kafka (only confirming when the summarisation is done and we've sent a response). To do this, we would have to significantly increase the confirmation timeout, which would result in larger batches of delivered messages, and it would delay our ability to detect failures. Remember the *smart endpoints, dumb pipes* tenet of microservices: we should not abuse Kafka to avoid having to implement smart endpoints. This means that each grouping microservice has to maintain its own persistent state that must enable it to replay the messages it has failed to process. Figure 1-10 reminds us where the summary service fits.

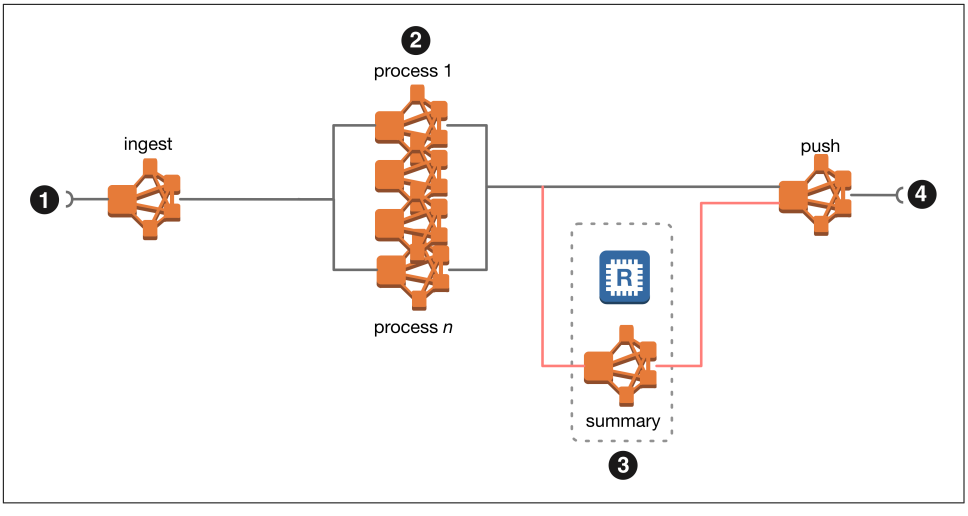

Figure 1-10. Summary service

Notice that the data store belongs to the summary microservice; the data in that store are not shared with any other microservice in the system. All that we have to resolve is exactly what data the service needs to persist to be able to recover from failures, and to maintain our message delivery guarantees. Let's imagine we only process messages for one transaction, and suppose that we need to consume 3 incoming messages before we can produce one summary response. Figure 1-11 shows that the microservice consumes one message at a time from Kafka.

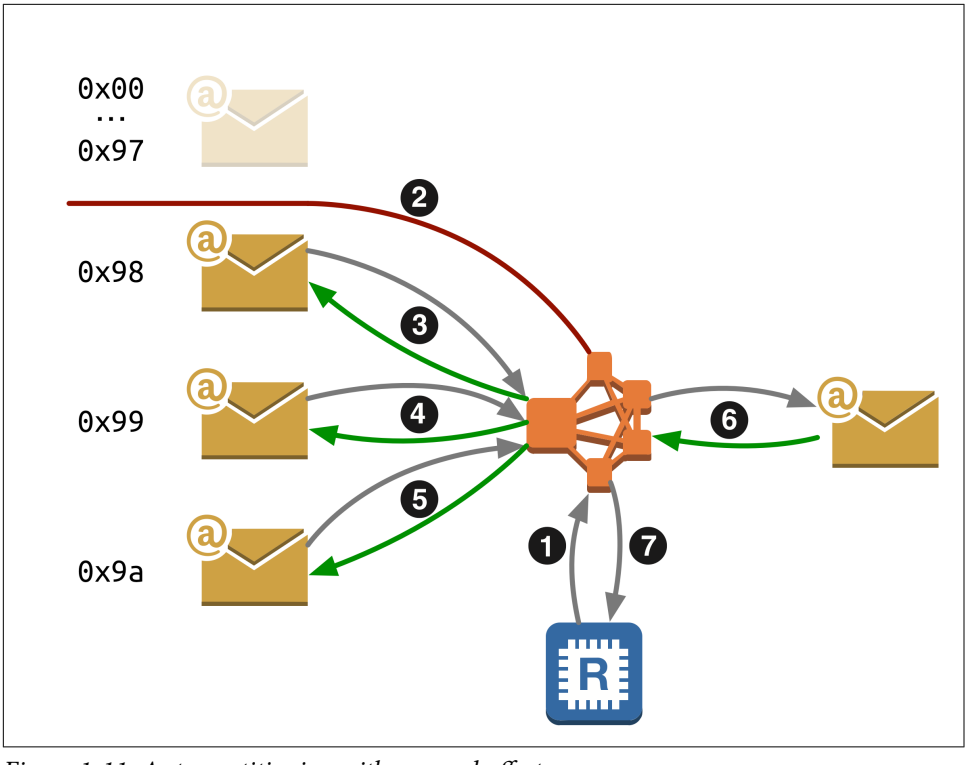

Figure 1-11. Auto partitioning with manual offsets

❶ When Kafka assigns the topic and partition to the listener in the summary microservice, the microservice loads the offsets where it wishes to start consuming messages from a database (`0x97`),

❷ Kafka then starts the subscription from the indicated offset,

❸ the microservice consumes a batch with one message starting at offset `0x98`; it immediately confirms the offset `0x98`: this informs Kafka that the consumer thread is healthy, and it delivers the next batch of messages to it,

❹ the microservice consumes a batch with one message starting at offset `0x99` and confirms the offset,

❺ the same flow happens for batch starting at `0x9a`, but the message—being the third number in the count of messages[8]—allows us to produce the summary message,

❻ the microservice delivers the summary message the output topic, and it receives a confirmation of successful delivery,

❼ the microservice writes the offset `0x9a` to its offset database.

This flow looks just like the event-sourcing we described in "Event-sourcing" on page 18. We are using Kafka as the event journal, so all that the summary microservice has to do is to remember is the offset that allows it to recover its state in case of failure. In order for the summary microservice to be as resilient as possible, it should tolerate as many failures in its dependencies as possible. Clearly, it cannot tolerate Kafka becoming inaccessible, but can it continue working or can it start if its offsets database is inaccessible? The answer dependes on how much re-processing the summary microservice is prepared to do if the offsets database is not accessible. If it is ready to reprocess all messages, it can completely ignore failures in writes and on failures of reads, it simply assumes zero offset. This sounds attractive, but if this microservice started processing messages from offset 0, it could generate load that could cause downstream microservices to fail; bringing a ripple of failures caused by extreme spike in messages. A better approach is to tolerate failures in offset writes for a certain period of time (during which we expect the offsets database to recover).

The summary state machine

The diagram in Figure 1-12 shows the in-flight states the transaction goes through; this will help us to implement the core of its logic.

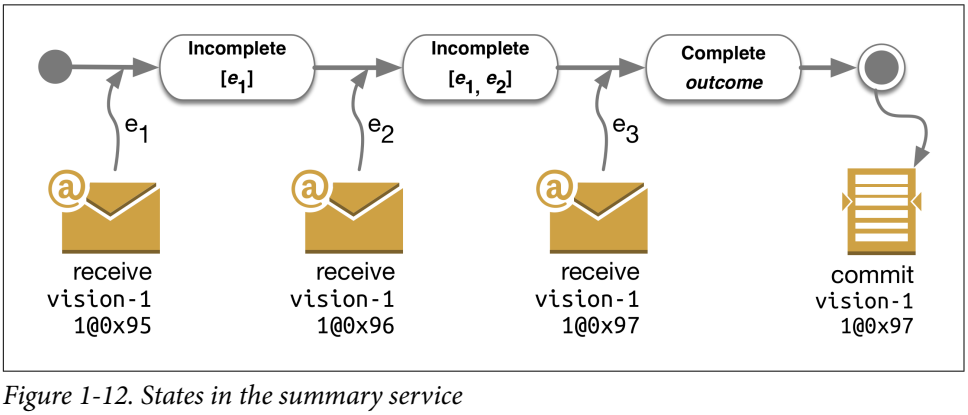

Figure 1-12. States in the summary service

8 The number three is key in defeating the Rabbit of Caerbannog using the Holy Hand Grenade of Antioch.

Example 1-19 shows the summarisation code, with the core state machine in the `Summary` trait and the `Summary.Incomplete` and `Summary.Complete` subtypes. We then define the `Summaries` case class that holds a collection of individual `Summary` instances, and applies the incoming Kafka messages (in the `ConsumerRecord` collection) to the `Summary` instances it holds, returning the new version of itself together with the completed outcomes and *offsets* of the topic partitions that can be saved to the microservice's database when the summary messages are successfully placed on the outgoing topic.

Example 1-19. Summary code

```
sealed trait Summary {
  def next(envelope: Envelope): Summary
}

object Summary {
  case class Incomplete private(events: List[Envelope]) extends Summary {
    override def next(envelope: Envelope): Summary = ...
  }

  case class Complete(outcome: Outcome) extends Summary {
    override def next(envelope: Envelope): Summary = this
  }
}

case class Summaries() {
  import Summaries._

  def appending(consumerRecords: List[ConsumerRecord[String, Envelope]]):
    (Summaries, Map[String, Outcome], Offsets) = ...
```

The state machine in code in Example 1-19 implements the logic of the `summary` service, but we're still missing the machinery that implements the event-sourcing service.

The summary actor

Completely asynchronous, event-driven implementation sounds daunting, but we use the actor toolkit Akka[akka] to handle the details of actor concurrency and to allow us to write ordinary Scala code. The core concept in Akka is *actor*, which is a container for [possibly mutable] state and behaviour, child actors, supervision strategy, and a mailbox. Messages (think instances of JVM classes) are placed in the mailbox by one set of threads, and are picked up and applied to the actor's behaviour by another set of threads. Should there be an exception during application of the actor's behaviour on the message from the mailbox, a supervision strategy determines how to resolve the problem. (The simplest case is to just restart the actor—that is, create a new instance of the actor—and continue delivering the messages from the mailbox to

this new instance.) Because Akka takes care of picking up messages from the actor's mailbox, it can guarantee that only one message at a time will be processed (though not necessarily in the order in which they were delivered to the mailbox!). Because of this, we can keep mutate the actor instance's state in the `receive` function (though we should not use `ThreadLocal` as there is no guarantee that the `receive` function will be applied to the messages from the mailbox from the same thread). Finally, each child actor falls into the parent actor's supervision strategy. Armed with this knowledge, we can tackle the code in Example 1-20.

Example 1-20. Summary code

```
class SummariesActor(consumerConf: KafkaConsumer.Conf[String, Envelope],
                     consumerActorConf: KafkaConsumerActor.Conf,
                     producerConf: KafkaProducer.Conf[String, Envelope],
                     redisClientPool: RedisClientPool,
                     topicNames: List[String]) extends Actor {

  private[this] val kafkaConsumerActor = context.actorOf(...) ❶
  private[this] val kafkaProducer = KafkaProducer(producerConf)
  private[this] var summaries: Summaries = Summaries.empty

  import scala.concurrent.duration._

  override def supervisorStrategy: SupervisorStrategy = ❷
    OneForOneStrategy(maxNrOfRetries = 3, withinTimeRange = 3.seconds) {
      case _ => SupervisorStrategy.Restart
    }

  override def preStart(): Unit = { ❸
    kafkaConsumerActor ! Subscribe.AutoPartitionWithManualOffset(topicNames)
  }

  override def receive: Receive = { ❹
    case AssignedListener(partitions: List[TopicPartition]) => ❺
      val offsetsMap = redisClientPool.withClient { ... }
      sender() ! Offsets(offsetsMap) ❻

    case SummariesActor.extractor(consumerRecords) => ❼
      val (ns, outcomes, offsets) = summaries.appending(consumerRecords.recordsList) ❽
      summaries = ns
      kafkaConsumerActor ! KafkaConsumerActor.Confirm(consumerRecords.offsets) ❾

      if (outcomes.nonEmpty) { ❿
        val sent = outcomes.map { case (transactionId, outcome) =>
          val out = Envelope(...)
          kafkaProducer.send(KafkaProducerRecord("summary-1", transactionId, out)) ⓫
        } ⓬
        import context.dispatcher
        Future.sequence(sent).onComplete { ⓭
          case Success(recordMetadata) => persistOffsets(offsets) ⓮
```

```
          case Failure(ex) => self ! Kill ⓯
        }
      }
    }

    private def persistOffsets(offsets: Offsets): Unit = { ⓰
      redisClientPool.withClient { ... }
    }

}
```

❶ To connect to Kafka, this actor creates a *child actor* that takes care of the Kafka connection and consumption, and delivers messages to this actor (the child actor's lifecycle and supervision strategy is bound to this actor),

❷ the supervisor strategy defines behaviour that Akka should take in case of failures in the actor's behaviour; in this case, we say that we want to re-create the instances of this actor in case of exceptions in `receive`, though up to three times in any period of three seconds,

❸ in the `preStart()` method, we *place a message* to the `kafkaConsumerActor`'s mailbox; the call completes as soon as the message is in the mailbox, but the `kafkaConsumerActor`'s behaviour will be triggered later when some other thread picks up the message from its mailbox and applies the `kafkaConsumerActor`'s behaviour to it,

❹ the actor's behaviour is a partial function implemented by pattern matching on the messages,

❺ one of the messages that this actor receives is the `AssignedListener` message, indicating that the Kafka client actor has established a connection,

❻ this actor needs to reply to it (`sender()`) with the offsets from which the message batch delivery should begin,

❼ the other messsage that this actor receives is the batch of messages from Kafka,

❽ the actor passes the batch of messages (`List[ConsumerRecord[String, Envelope]]`) to the `summaries.appending` function, which returns a new version of `summaries`, together with the completed summaries and the offsets where subscription needs to start from should this actor fail in order not to lose any message,

❾ immediately after updating our state, the actor confirms delivery of the offsets to Kafka,

❿ if the new batch of messages resulted in some transactions completing,

⓫ the actor places a message (constructed from the outcome) on the outgoing topic,

⓬ the result is a collection of futures of delivery confirmation (`Seq[Future[RecordMetadata]]` in the Scala API); when all these futures succeed, we will know that the summary messages are indeed published,

⓭ to do so, we have to *flip the containers*: turn `Seq[Future[RecordMetadata]]` into `Future[Seq[RecordMetadata]]` using the `Future.sequence` call,

⓮ when all the futures in the sequence succeed, the actor can persist the offsets of messages that were used to compute the summaries into this microservice's offsets database (if the actor crashes now, the messages are published!),

⓯ should any of the futures in the sequence fail, we trigger the actor's supervision strategy by sending it the `Kill` message (we are also not persisting the offsets of the messages that were used to compute the summaries: we failed to publish the results and we will be restarted, so we need to re-process the messages again!),

⓰ finally, the implementation of the `persistOffsets` function should be lenient with respect to the failures of the offsets database (it can, for example, only trigger the actor's supervision strategy if there are 10 consecutive failures).

Akka allows us to maintain mutable state in an actor-based concurrency system; this fits well into the message-based nature of the system we are building. The supervision strategy in Akka allows us to define what should happen in case of failures. It is important to remember that the supervision strategy can be applied at any level; the supervision strategy at the root level (i.e. actor without any parent) is to terminate the actor system. And so, the summary microservice can recover by attempting to recover the component that is closest to the failure, propagating up to the JVM process level if the component-level recovery is not successful. Once the JVM exits with a non-zero exit code, causing the container in which it runs to exit with the same non-zero exit code, the system's distributed systems kernel should attempt recovery by restarting the container. Contrast this with heavy-handed recovery, where a container hosting the application exits after the first exception, because there is no hierarchy of components within it; which causes additional work for the distributed systems kernel, Kafka, and Zookeeper.

Tooling

Each member of the development team can clone the project and build any of the microservices. However, it would be difficult for one engineer to build every microservice that the system needs, and to configure its dependencies; though it is something that the engineer might need to work on one of the microservices. The situation becomes even more complex when the engineer needs to run large-scale testing and training, which is particularly applicable to computer vision & machine learning.

Machine learning training and the appropriate testing is taking us away from entirely deterministic world, but that is nowhere near the unpredictability of *chaos engineering*. In chaos engineering, we construct tests that verify the system's behavior while we inject faults into the running system. The faults can be as obvious as complete node failures and network partitions; more subtle faults triggered by garbage-collection pauses and increased I/O latencies and packet drops; all the way to very subtle faults such as successful system calls that nevertheless yield the wrong result—think socket or file read operations that indicate successful result, but that fill the userspace buffers with the wrong bytes; or concurrency problems exposed by the [hardware] power and performance management running different cores of modern CPUs at different speeds.

To have confidence in the stability of a distributed microservices system, we must verify that the system produces expected results *under load* and *in presence of chaos*.

Distributed systems might exhibit unexpected emergent behaviour patterns under stress—think failures caused by extreme spikes in the load, causing further cascading failures. Production environment is not the best place to first observe unexpected emergent behaviour.] The image processing system we describe here is tested under load and chaos in the testing environment, but the production environment only includes slight, but continuous load test. (The load test represents approximately 10 % of the total load.)

Development tooling

Let's start the discussion of tooling from development tooling. It must be convenient and reliable to use in all stages of the development and deployment pipeline: all the way from engineers working on their machines to the continuous delivery infrastructure deploying production artefacts. The motivation is that it is likely that each development machine is set up differently, which might result in the build process producing a different artefact. A good example is where the engineers use mac OS for development, producing Mach-O x86_64 binary linking the mac OS shared objects, but such binary does not run in a Docker container. Docker runs Linux, so the binaries need to be a [x86_64] ELF binary linking the Linux shared objects. The development tooling must simplify and streamline development tasks that are difficult to

achieve without complex set up; a good example is computer vision & machine learning training and validation.

Let's consider the development tooling for the native code (tooling for JVM-based code is similar). We have two Docker containers: the runtime container, which includes dependencies (shared objects) needed to run the native code, and the development container, which includes the build tools and the development versions of the dependencies. We initially started by building the runtime container first, reasoning that we will use the runtime image as base and then install the development tooling on top of it. However, we found that some of the dependencies we were using we had to build from sources (the versions available in the package manager were not recent enough), forcing us to reverse the process. We build the development container first, then remove the development packages to create the runtime image.

Example 1-21. Development image

```
FROM ubuntu:16.04

ENV CUDA_VERSION 8.0
ENV CUDA_PKG_VERSION 8-0=8.0.44-1

RUN apt-get update && apt-get install -y --no-install-recommends \
        cuda-core-$CUDA_PKG_VERSION \
        ...
        git \
        g++ \
        cmake \
        && \
    ln -s cuda-$CUDA_VERSION /usr/local/cuda && \
    rm -rf /var/lib/apt/lists/*
...

RUN curl -L -o opencv3.zip https://github.com/opencv/opencv/archive/3.2.0.zip && \
    unzip opencv3.zip && \
    ...
    cmake .. && \
    make -j8 install

RUN echo "/usr/local/cuda/lib64" >> /etc/ld.so.conf.d/cuda.conf && \
    ldconfig
```

Running `docker build -t oreilly/rac-native-devel .` in a directory containing —among simple build shell scripts—the `Dockerfile` from Example 1-21 builds the development container with the dependencies that can build all our native / computer vision microservices. To build the runtime container, we take the development container and remove the development packages.

Example 1-22. Development image

```
FROM oreilly/rac-native-devel:latest

RUN apt-get update && apt-get remove -y --no-install-recommends \
        git \
        gcc \
        g++ \
        cmake \
        && \
      ln -s cuda-$CUDA_VERSION /usr/local/cuda && \
      rm -rf /var/lib/apt/lists/*

...
```

Unfortunately, this results in very large runtime image, which we solve by squashing the runtime image using docker-squash[docker-squash]. This gave us acceptable runtime image sizes for both the native / CUDA containers as well as for the JVM containers (450 MiB and 320 MiB, respectively). The result is that the engineers can clone the appropriate sources and then use the development container to build & test their code independently of their machine setup. To further help with convenience, we have a `racc` script, which detects the nature of the project (native or JVM), and uses a parameter to decide which tooling container to run.

Example 1-23. Building using the development tooling

```
~/ips/faceextract $ racc build
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
...
[ 93%] Building CXX object rapidcheck-build/.../detail/Recipe.cpp.o
[ 94%] Building CXX object rapidcheck-build/.../detail/ScaleInteger.cpp.o
[ 96%] Linking CXX executable main
[ 96%] Built target main
[ 98%] Linking CXX static library librapidcheck.a
[ 98%] Built target rapidcheck
Scanning dependencies of target main-test
[100%] Linking CXX executable main-test
[100%] Built target main-test
Test project /var/src/module/target
    Start 1: all
1/2 Test #1: all ..............................   Passed    0.01 sec
    Start 2: all
2/2 Test #2: all ..............................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.12 sec
```

The result is a collection of built and tested artifacts (executables, shared and static libraries, test and environment reports), which can be given to the next set of containers, which implement the testing pipeline (performance and chaos testing, large-scale vision evaluation), and the inventory discovery.

Summary

In this chapter, we have described the architecture of an image processing system that ingests a large volume of images from untrustworthy devices (mobiles, IoT cameras).

The system only trusts its own code, so even if the edge device has sufficient power to perform *all* required computer vision tasks, the system repeats the work once it ingests the raw data. The services in the system take great care not to lose any messages by using message broker that can distribute the messaging load into partitions and provide data loss protection through replication. The services rely on event-sourcing; either implicitly by having the broker provide the journalling and offset snapshots; by leaving the message journal in the broker, but maintaining their own offset snapshots; or by maintaining their own journal and offset snapshot stores.

As we have shown here, it is important for the services not to shy away from maintaining state where applicable. A system that can recall information, should recognize services that contain that state. (Compare that with a system that comprises services that are stateless in the sense that they do not contain in-flight state, but rely instead on a [distributed] database.)

You learned about the importance of very loose coupling between isolated services, but the strict protocols that such de-coupling and isolation requires. Reactive microservices systems are ready for failures, not simply by being able to retry the failing operations, but also by implementing graceful service degradation. The technical choices give the business the opportunity to be creative in deciding what constitutes a degraded service. (Recall that the authorization service degrades by granting more permissions!) We highlighted the absolute requirement for the services to be responsive, even if the response is rejection; we learned that back-pressure is the key to the system as a whole taking on only as much work as its services can process. We briefly tackled the importance of good development tooling, especially when it comes to native services.

- [boner2016] Jonas Bonér. (2016). *Reactive Microservices*. OReilly. ISBN 978-1-491-95779-0.
- [spolsky2002] Joel Spolsky. (2003). *The Iceberg Secret, Revealed*. Retrieved 23 January 2017 from *https://www.joelonsoftware.com/2002/02/13/the-iceberg-secret-revealed/*.
- [scalacheck] ScalaCheck. (2016). *ScalaCheck: Property-based testing for Scala*. Retrieved 20 February 2017 from *https://www.scalacheck.org/*.
- [rapidcheck] rapidcheck. (2016). *QuickCheck clone for C++ with the goal of being simple to use with as little boilerplate as possible*. Retrieved 10 January 2017 from *https://github.com/emil-e/rapidcheck*.
- [protobuf] Google. (2017). *Protocol Buffers*. Retrieved 11 January 2017 from *https://developers.google.com/protocol-buffers/*.
- [gtest] Google. (2017). *Google Test*. Retrieved 4 January 2017 from *https://github.com/google/googletest*.
- [librdkafka] *The Apache Kafka C/C++ library*. (2017). Retrieved 15 January 2017 from *https://github.com/edenhill/librdkafka*.
- [docker-squash] docker-squash. (2017). *Docker image squashing tool*. Retrieved 10 February 2017 from *https://github.com/goldmann/docker-squash*.
- [sumologic] sumologic. (2017). *Real-time Application Monitoring and Trouble-shooting*. Retrieved 20 February 2017 from *https://www.sumologic.com/use-cases/it-operations/*.
- [pingdom] pingdom. (2017). *Website monitoring for everyone*. Retrieved 20 February 2017 from *https://www.pingdom.com/*.
- [consul] Consul by HashiCorp. (2017). *Service discovery and configuration made easy.* Retrieved 20 February 2017 from *https://www.consul.io/*.
- [kinesis] Amazon Web Services. (2017). *Amazon Kinesis*. Retrieved 20 February 2017 from *https://aws.amazon.com/kinesis/*.