O'REILLY®

Compliments of Lightbend

Deploying Reactive Microservices

Strategies and Tools for Delivering Resilient Systems

Edward Callahan

Deploying Reactive Microservices

Strategies and Tools for Delivering Resilient Systems

Edward Callahan

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

Deploying Reactive Microservices
by Edward Callahan

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
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2017: First Edition

Revision History for the First Edition
2017-07-06: First Release

978-1-491-98148-1

[LSI]

Table of Contents

CHAPTER 1
Introduction

> Every business out there now is a software company, is a digital company.
>
> —Satya Nadella, *Ignite 2015*

This report is about deploying Reactive microservices and is the final installment in this Reactive microservices series. Jonas Bonér introduces us to Reactive and why the Reactive principles so inherently apply to microservices in Reactive Microservices Architecture. Markus Eisele's Developing Reactive Microservices explores the implementation of Reactive microservices using the Lagom Framework. You're encouraged to review those works prior to reading this publication. I will presume basic familiarity with Reactive and the Reactive Manifesto.

Thus far in the series, you have seen how adherence to the core Reactive traits is critical to building services that are decoupled but integrated, isolated but composable, extensible and maintainable, all while being resilient and scalable in production. Your deployment systems are no different. All applications are now distributed systems, and distributed applications need to be deployed to systems that are equally designed for and capable of distributed operation. At the same time, the deployment pipeline and cluster can inadvertently lock applications into container-specific solutions or services. An application that is tightly coupled with its deployment requires more effort to be migrated to another deployment system and thus is more vulnerable to difficulties with the selected provider.

This report aims to demonstrate that not only should you be certain to utilize the Reactive patterns in our operational platforms as well as your applications, but in doing so, you can enable teams to deliver software with precision and confidence. It is critical that these tools be dependable, but it is equally important that they also be enjoyable to work with in order to enable adoption by both developers and operations. The deployment toolset must be a reliable engine, for it is at the heart of iterative software delivery.

This report deploys the Chirper Lagom sample application using the Lightbend Enterprise Suite. The Lightbend Enterprise Suite provides advanced, out-of-the-box tools to help you build, manage, and monitor microservices. These tools are themselves Reactive applications. They were designed and developed using the very Reactive traits and principles examined in this series. Collectively, this series describes how organizations design, build, deploy, and manage software at scale in the data-fueled race of today's marketplace with agility and confidence using Reactive microservices.

Every Company Is a Software Company

Change is at the heart of the drive to adopt microservices. Big data is no longer at rest. It is now fast data streams. Enterprises are evolving to use fast data streams in order to mitigate the risk of being disrupted by small, faster fish. They are becoming software service providers. They are using software and data for everything from enhancing user experiences to obtaining levels of efficiency that were previously unimaginable. Markets are changing as a result. Companies today increasingly view themselves as having become software companies with expertise in their traditional sectors.

In response, enterprises are adopting what you would recognize as modern development practices across the organization. They are embracing Agile and DevOps style practices. The classical centralized infrastructure solutions are no longer sufficient. At the same time, organizations now outsource their hardware needs nearly as readily as electrical power generation simply because it is more efficient in most every case. Organizations are restructuring into results-oriented teams. Product delivery teams are being tasked with the responsibility for the overall success of services. These forces are at the core of the rise of DevOps practices and the adoption of deployment platforms such as Lightbend Enterprise Suite, Kuber-

netes, Mesosphere DC/OS, IBM OpenWhisk, and Amazon Web Services' Lambda within enterprises today.

Operations departments within organizations are increasingly becoming a resource provider that provisions and monitors computing resources and services of various forms. Their focus is shifting to the security, reliability, resilience, and efficient use of the resources consumed by the organization. Those resources themselves are configured by software and delivered as services using very little or no human effort.

Having been tasked to satisfy many diverse needs and concerns, operation departments realize that they must modernize, but are understandably hesitant to commit to an early leader. Consider the *serverless*, event-driven, Function as a Service platforms that are gaining popularity for their simplicity. Like the batch schedulers before them, many of these systems will prove too limited for system and service use cases which require a richer set of interfaces for managing long-running components and state. Operations teams must also consider the amount of vendor lock-in introduced in the vendor-specific formats and processes. Should the organizations not yet fully trust cloud services, they may require an on-premise container management solution. Building one's own solution, however, has another version of lock-in: owning that solution. These conflicting interests alone can make finding a suitable system challenging for any organization.

At the same time, developers are increasingly becoming responsible for the overall success of applications in deployment. "It works for us" is no longer an acceptable response to problem reports. Development teams need to design, develop, and test in an environment similar to production from the beginning. Multi-instance testing in a clustered environment is not a task prior to shipping, it is how services are built and tested. Testing with three or more instances must be performed during development, as that approach is much more likely to detect problems in distributed systems than testing only with single instances.

Once confronted with the operational tooling generally available, developers are frustrated and dismayed. Integration is often cumbersome on the development process. Developers don't want to spend a lot of time setting up and running test environments. If something is too difficult to test and that test is not automated, the

reality is too often that it just won't be properly tested. Technical leads know that composable interfaces are key for productivity, and that concurrency, latency, and scalability can cripple applications when sound architectural principles are not adhered to. Development and operations teams are demanding more from the operational machinery on which they depend for the success of their applications and services.

Microservices are one of the most interesting beneficiaries of the Reactive principles in recent years. Reactive deployment systems leverage those principles to meet today's challenges of cloud computing, mobile devices, and Internet of Things (IoT).

Full-Stack Reactive

Reactive microservices must be deployed to a Reactive service orchestration layer in order to be highly available. The Reactive principles, as defined by the Reactive Manifesto, are the very foundation of this Reactive microservices series. In Reactive Microservices Architecture, Jonas explains why principles such as acting autonomously, Asynchronous Message-Passing, and patterns like shared nothing architecture are requirements for computing today. Without the decoupling these provide, it is impossible to reach the level of compartmentalization and containment needed for isolation and resilience.

Just as a high-rise tower depends upon its foundation for stability, Reactive microservices must be deployed to a Reactive deployment system so that organizations building these microservices can get the most out of them. You would seriously question the architect who suggests building your new high-rise tower on an existing foundation, as is. It may have been fine for the smaller structure, but it is unlikely to be able to meet the weight, electrical, water, and safety requirements of the new, taller structure. Likewise, you want to use the best, purpose-built foundation when deploying your Reactive microservices.

This report walks through the deployment of a sample Reactive microservices-based application using the Developer Sandbox from Lightbend Enterprise Suite, Lightbend's offering for organizations building, managing, and monitoring Reactive microservices. The example application is built using Lagom, a framework that helps

Java and Scala developers easily follow the described requirements for building distributed, Reactive systems.

Deploy with Confidence

A deployment platform must be developer and operator friendly in order to enable the highly productive, iterative development being sought by enterprises undergoing software-led transformations. Development teams are increasingly realizing that their Reactive applications should be deployed to an equally Reactive deployment platform. This increases the overall resilience of the deployment while providing first-class support for peer clustering applications such as Actor systems. With the complexity of managing state in a distributed deployment being handled Reactively, the deployment workflow becomes a simplified and reliable pipeline. This frees developers to address business needs instead of the many details of delivering clustered services.

The next chapter examines the importance of the Reactive traits in building a microservices delivery solution. We'll look at key usability features to look for in a Reactive deployment system. In Chapter 3 you will test an implementation of these characteristics applied in practice by deploying the Chirper Lagom sample application using Lightbend Enterprise Suite. We'll explore the resilience of the system by inducing failures and watching as the system responds and self-heals. I will then close out this Reactive microservices series and allow you to continue enjoying the thrill of a fully Reactive microservices stack deployment!

CHAPTER 2
The Reactive Deployment

> Failure is always an option; in large-scale data management systems, it is practically a certainty.
>
> —Alvaro, Rosen, and Hellerstein, *Lineage-driven Fault Injection*

The way applications are deployed is changing just as rapidly as the development tools and processes being used to produce those applications. Microservices are deployed as systems to fleets of nameless *cattle* servers. Unlike a set of named *pet* hosts that you care for and upgrade, *cattle* are immutable and replaceable. System security updates? New kernel? No problem. Introduce new instances with updates to the cluster fleet. Workload is migrated off the older, unpatched instances to the newly minted ones. The outdated nodes are terminated once idled of all executions.

The physical world into which you deploy our applications, however, hasn't changed much by comparison. Hardware fails. Mean time before failure maybe longer, but mechanical failure is still inevitable. Processes will still die for numerous reasons. Networks can and will partition. Failure cannot be avoided. You must, instead, embrace failure and seek to keep your services available despite failure, even if this requires operating in a degraded manner. Let it crash! Your systems must be capable of surviving failures. Instead of attempting to repair nodes when they fail, you replace the failing resources with new ones.

Consider Chaos Monkey, a service that randomly terminates services in applications to continuously test the system's ability to recover. Netflix runs this service against its production environ-

ment. Why? As stated in the readme: "Even if you are confident that your architecture can tolerate a system failure, are you sure it will still be able to next week, how about next month?"

Persistent data storage is required in any application that handles business transactions. It is also more complicated than working with stateless services. Here as well, our Reactive principles help simplify the solution. Event sourcing and CQRS isolate backend data storage and streaming to engines like Apache Cassandra and Apache Kafka. Their durable storage needs are likewise isolated. This can be done using roles to direct those services to designated nodes, or by using a specialized service cluster to provide the storage engine "as a service." If using specialized nodes, those nodes and the services they execute can have a different life cycle than that of stateless services. Shards need time to synchronize, volumes need to be mounted, and caches populated. Cluster roles enable application configuration to specify the roles required of a node that is to execute the service. Specialized clusters make persistence issues the concern of the service provider. That could be Amazon Kinesis or an in-house Cassandra team providing the organization with Cassandra as a service. The storage as a solution approach offers the benefit that the many details of persistence are the provider's problem.

Tomorrow's upgrades require semantic versioning today for the smooth managing of compatibility. Incompatible, major version upgrades use just-in-time record migration patterns instead of big bang style, all-in-one migrations. Minor version, compatible upgrades are rolled in as usual. Applications must be able to express compatibility using system and version number declarations. Simple string version tags lack the semantics needed to automatically determine compatibility, limiting autonomy of the cluster services. During an upgrade, API gateways and other anti-corruption layers can operate with both service versions simultaneously during the transition. This enables you to better control the migration to the new version. Schema-incompatible upgrades can be further controlled with schema upgrade-only releases or by using new keyspaces. Either approach can be used to ensure there is always a rollback path should the upgrade fail.

The Reactive deployment uses the Reactive principles to embrace failure and to be resilient to failure. With a fully Reactive stack deployment, you enable confidence. Immutability provides the ability to roll back to known good states. Confidence and usability

enable teams to deliver what would otherwise be very difficult. This chapter will examine the features you should expect from deployment tooling today.

Distributed by Design

First and foremost, your deployment platform must be a Reactive one. A highly available application should be deployed to a resilient deployment platform if it itself is to be highly available. The reality is that systems are either well designed for distributed operation or are forever struggling to work around those realities. (In the physical world, the speed of light is the speed limit. It doesn't matter what type of cable you run between data centers, the longer the cable between the two ends, the longer it takes to send any message across the cable.)

The implications of your services failing and not being available are wide reaching. System outages and other software application–caused disruptions are part of daily news cycles. On the other end of the spectrum, consider the user experience when using old, slow, and other aged systems. Like a blocking writer in data stream processing, you immediately notice the impact. If you need to make multiple updates into a system that requires you to perform one change at a time, you may reconsider how many changes you *really* need. If the system further encumbers you with wait periods, refusing to input your next update until all writers have synchronized, making many changes quickly becomes an exercise in patience. Even if you discount these as inconveniences to be tolerated, you cannot deny their impact on productivity. The experience is boring, if not outright demotivating. If allowed, you become more likely to accept "good enough" solely to avoid another agonizing experience of applying those updates. You avoid interacting with the system.

Distributed system operation is difficult. In describing the architecture of Amazon's Elastic Container Service (ECS), Werner Vogel notes the use of a "Paxos-based transactional journal data store" to provide reliable state management. Docker Engine, when in swarm mode, uses a Raft Consensus Algorithm to manage cluster state. Neither algorithm is known for its simplicity. The designers felt these components were required to meet the challenges of distributed operation.

The Lightbend Enterprise Suite's Reactive Service Orchestration feature is a masterless system. Conflict-free replicated data types, or CRDTs, are used for reliably propagating state within the cluster, even in the face of network failure. Everything from available agent nodes to the location of service instance executions is shared across all members using these CRDTs. This available/partition tolerance–based eventual consistency enables coordination of data changes in a scalable and resilient fashion.

The Benefits of Reliability

> Fear is the mind-killer.
>
> —Frank Herbert, *Dune*

Users must be able to deploy with confidence. Teams must be able to deploy updates with the comfort of knowing that although they may need to roll back to the previous version, they will be able to do so relatively easily. They will always have a path back to the last-known good configuration. Should the release fail for any reason, they simply revert to the previous version. Loading, scaling, and stopping services requires push-button simplicity. Top-level choices are go forward to the next release or go back to the previous release. Users should not be fearful of rolling out a new feature. Without confidence in the delivery mechanism and its ability to return to a known good state, a team may hesitate and miss important opportunities.

Consider the experience of using a well-designed application. It provides comfort in the knowledge that you should not be able to unintentionally harm yourself. If you are about to accidentally delete something important, the system might prompt you for confirmation or require the owner account password to be entered. This encourages you to explore the interface, which frees you to discover new features. The virtuous cycle continues as confidence in the interface makes you more likely to try the new feature. What if your development team approached deployment with the trivial amount of anxiety that you feel when using a vending machine's currency reader? If the desired outcome isn't realized, the machine spits the currency back out, but the team is otherwise none the worse for the experience. Deploying a new release should be equally mundane. Every time.

The critical importance of developer velocity, the rate at which features can be delivered, is well understood by Netflix. In a blog post

regarding its evolution of container usage, Netflix directly attributes speed and ease of experimental testing to the ability to "deploy to production with greater confidence than before [containers]." Furthermore, "this velocity drives how fast features can be delivered to Netflix customers and therefore is a key reason why containers are so important to our business."

Good clustering and scheduling systems empower their users. Organizations are challenging teams to be even more imaginative, to ask what could be if failure was not an concern. From easy-to-use developer sandboxes for safe experimentation to appliance-like simplicity for delivery and rollback, teams need tools that support rapid *what if* innovation cycles required to answer that question. As production software delivery becomes more critical to the success of enterprises, the benefit and value of a reliable deployment system that is easy to use becomes quite clear. Waiting until Monday to respond is no longer good enough.

Traits of a Reactive Deployment

It is easy to see that the core Reactive attributes—responsive, resilient, elastic, and message-driven—are desirable in a distributed deployment tool system. What does this mean in practice? What does it look like? More importantly, what advantages can it afford us? Eventual consistency, event sourcing, and other distributed patterns can seem foreign to our normal usage at first. In reality, you are likely already using eventually consistent systems in many of the cloud services you currently consume. The following sections discuss characteristics to consider when choosing a deployment system.

Developer Friendly

A deployment control system should enable teams to push new ideas out quickly and easily. It should nurture creativity, not inhibit it. The greater a team's velocity, the faster the team can realize its vision. It should be straightforward to package, deploy, and run a service in the cluster environment, be it locally or in the cloud. A deployment system must be built from the ground up with the developer's needs being considered.

Developer friendly means allowing developers to focus on the business end of the application instead of on how to build packages, find

peers, resolve other services, and access secrets. Security and network partition detection alone can easily become significant undertakings when building your own solution. In particular, a developer-friendly deployment system should:

- Be simple to test services in a local machine cluster before merging
- Support Continuous Integration and Continuous Delivery (CI/CD) to test or staging environments
- Provide application-level consolidated logging and event viewing
- Be composable so that you can manage your services as a fleet instead of herding cats
- Have cluster-friendly libraries and utilities to keep deployment-specific concerns out of your application code. Examples include:
  - Peer node discovery with mutual authentication
  - Service lookup with fallbacks for dev and test environments
  - User quota, mutual service authentication, secret distribution, config checker, diagnostics recorder, and assorted helper services

Ease of testing

It must be simple for developers to test locally in an environment that is highly consistent with production. Testing is fundamental to the deployment process. Users must be able to test at all stages in an appropriate production-like environment and do so easily. Hosted services and other black-box systems can be difficult to mock in development and generally require full duplicate deployments for the most basic of integration testing.

For developers, particularly those accustomed to using language platforms that do not provide dependency management, Docker makes it simple to quickly test changes in the containerized environment. Consider a typical single-service application that can be run in place out of the source tree for development run and test such as a common blog or wiki app. Setting up the host environment for testing changes can require more effort than the changes themselves. Virtual Machines (VMs) help, but are big, heavyweight

objects better suited for less dynamic, lab-style environments. It still takes minutes to launch a VM from start. That is no longer fast enough. VMs are also very difficult to share, such as by attachment in a bug report. Containers provided us with operating system–level virtualization that is much more transportable. Like microservices, containers mostly have a single purpose.

An important decision early in the life of a software project is the choice of packaging. It should be easy to produce the bundle of all the objects needed to run your service in the cluster. This will include your container image definition, such as the Dockerfile, container metadata, dependencies, and any other binaries required to execute the container. Being able to run the container bundle directly in a container engine is good, but it doesn't assure us that the service can start, locate other services, or otherwise function in the production cluster. You must be able to validate both the container image and the cluster system bundling so that you don't spend cluster resources troubleshooting packaging issues.

You need to easily be able to test deploy your changes in a local developer sandbox that is highly consistent with the production deployment *before* submitting your changes as a Pull Request (PR). You need to be confident that you have correctly bundled your service for scheduling in the cluster. Creating tests and setting up Continuous Integration (CI) to run them continuously is fundamental to practices like Test-Driven Development. Your CI tests should likewise be able to validate the bundled service using the developer sandbox environment.

Continuous Delivery

A workflow-driven Continuous Delivery (CD) pipeline from development to production staging is a foundational part of any software project. A reliable, easy-to-use CD pipeline is not only an important stabilizer to the project, it is key to enabling innovative iteration. After developers submit their PRs, CI will test the proposed reversion. CI also uses the developer sandbox version of the cluster to test the changes. Once accepted and merged, the update is deployed. This will be as staging or test instances to the production cluster, or sometimes to a dedicated test cluster with a test framework such as Gatling.io running against it to validate performance under load. For most teams this means that every time there is a new head revision of the release branch, it is delivered to a cluster in a pre-

production configuration once all tests and checks pass. Other projects will be deployed directly to production, particularly those with sufficient test coverage to have nearly no risk.

Publication of a revision to production is then a simple matter of "promoting" the desired revision from staging to production. Promotion is the process of deploying the specified revision's bundle package with the current production configuration. However, not any old bundle in the repository is available for publication to production. Only those builds that were successful in the entire CD process are available for promotion. The initial delivery of a new version to take live traffic is often limited to a single instance at first. This first *canary* instance is intensely monitored for any anomalies and new or increased errors before migrating the entire production load to the new version.

Like other failures, you must accept and embrace the need to roll back a deployment. It is not an exception, it is plan *B*. When user experiences are being impacted, or service levels are otherwise failing due to a release, you quickly revert to the previous known good version. Then you can reevaluate and try again. For stateless and compatible service upgrades, this can be readily achieved by leaving the last deployed version loaded but not running in the cluster. For major upgrades or more complicated cases, you will often shift load between the two active applications at a proxy or routing layer. Regardless of how you migrate requests, the delivery pipeline only goes forward. You never want to need to hurriedly deploy a PR to revert the bad commit. You simply restart the old version if needed and re-shift load back. Once you've determined what went wrong, you deploy a new PR into the pipeline.

Secrets such as tokens, private keys, and passwords must be encrypted and their access strictly controlled. The service code should never contain any configuration values beyond the default values required for running unit tests. As stated by the Twelve-Factor App, a popular methodology regarding building services: "A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials." The application project code will often contain developer default secret values. They are overridden and supplemented with the correct values for the target environment at deployment. The secrets in the code have no value beyond development testing. Secrets must be delivered to the appli-

cation securely and never stored or transmitted in cleartext. The distribution of secrets must be a trusted service using mutual authentication with access logging. Such services are complicated and easy to get wrong. Look for integrations with proven solutions, such as Vault or KeyWhiz. The desired result is that you never modify the application service bundle package produced by the delivery pipeline. Ever. Instead, operators pair the application bundle with the appropriate secrets using the container cluster management system and its secrets distributions. In the case of the CD pipeline, the new versions are delivered to the cluster using staging or similar preproduction test credentials. Operators simply redeploy the verified and tested service bundle with the production secrets. Only authorized operators have access to the production secrets. They need not even know nor see the actual secret. They only need access to it in order to deploy with it. Thus the application can always be distributed, tested, and iterated without compromising any credentials or other sensitive information.

Cluster conveniences

You want your teams to focus on addressing business needs, not managing cluster membership, security, service lookup, and many other moving parts. You will want libraries to provide helper functions and types for dealing with the common tasks in your primary languages, with REST and environmental variables for the other needs. Good library and tool support may seem like conveniences for lazy developers, but in reality they are optimizations that keep the cluster concerns out of your services so your teams can focus on their services.

Service Discovery, introduced in Reactive Microservices Architecture, is an essential part of a microservices-based platform. Eventually consistent, peer gossip-based service registries are used for the same reason strong consistency is avoided in your application services: because strong consistency comes at a cost and is avoidable in many scenarios. Library support should include fallbacks for testing outside of the clustering system. Other interstitial concerns include mutual service authentication and peer-node discovery. If it is too difficult to encrypt data streams that should be encrypted, they are more likely to be unencrypted, or worse, not encrypted properly.

User quotas, or request rate limits, are a key part of keeping services available by preventing abuse, intended or otherwise. A user-

friendly deployment system prevents users from making mistakes. You want to able to install and manage all the services of an application as a single unit. This enables easier integration testing and allows for wider participation in the success of an application. How microservices form an application is a development concern. Otherwise, you are delivering a loose-bag collection of services "Ikea style" —some assembly required. Frameworks can have more options and choices than Starbucks offers in its coffee drinks. It is too easy for even the most experienced developer to overlook a problem. Configuration review utilities, such as the Akka configuration checker, can avoid costly time-consuming mistakes and performance-killing mismatches.

Composability

You want a descriptive approach that enables you to treat your infrastructure as code, and apply the same techniques you apply to application code. You want to be able to pipe the output of one command to another to create logical units of work. You want composability.

Composability is no accident. It generally requires a well-implemented domain-driven design. It also requires real-world usage: teams building solutions, overcoming obstacles, and enhancing and fixing the user interfaces. When realized, "composability enables incremental consumption or progressive discovery of new concepts, tools and services." Incremental consumption complements the "just the right size" approach to Reactive microservices.

Operations Friendly

Operations teams also enjoy the benefits of the developer-friendly features I noted. Meaningful application-specific data streams, such as logging output and scheduling events, benefit all maintainers of an application. Accounting only for its service provider and reliability roles, operations has many needs beyond those of the developers.

A fundamental aspect of any deployment is where it will reside, on which physical resources. Operations must integrate with both the new *and* existing infrastructure while enforcing business rules and best practices. Hybrid cloud solutions seek to augment on-premise resources with cloud infrastructure. The latency introduced between on-premise and cloud resources makes it difficult to scale a single

application across locations. The cumulative response times are just too long for servicing human-initiated requests.

Vendor lock-in remains a concern for many developers, and for good reason. At the same time, cloud service vendors seek to create stickiness in their services, for obvious reasons. Services are defined and managed as containers, but data persistence, load balancing, peer networking, secrets, and the surrounding environmental needs often are handled most easily when consuming the cluster vendors' commercial add-ons. This can force teams to choose between adopting the ready-made, vendor-specific solutions or building out their own, more portable solution. Some teams will decide that they cannot possibly take any path but the most expedient one. They accept that the overall project will be difficult if not impossible to move. Like Cloud Foundry, OpenShift, Heroku, and other Platform as a Service vendors, the more tightly the application is integrated into the stack, the more complexity will need to be handled in order to break that dependency.

Today, many are choosing to mitigate these risks with systems like the Lightbend Enterprise Suite, DC/OS, Docker Swarm, and Kubernetes. By consuming only basic infrastructure and utilizing industry standards, organizations can better abstract across multiple clouds, including those utilizing existing, on-premise, data centers. Even when you use multiple types of clusters across regions, divisions, customers, etc, you can still have a single deployment target to package and test for. DevOps tooling, such as Terraform and Ansible, further isolate teams from vendor specifics, much like printer drives save operating systems from needing device-specific knowledge for any printer a user might want to use.

Lightbend Enterprise Suite's Reactive Service Orchestration feature, part of its Application Management features, is packaged and delivered as ConductR. ConductR offers an additional option with the ability to run either standalone or within a scheduled cluster, currently DC/OS. When Reactive microservices are deployed using ConductR, the cluster itself can be running directly on x64 Linux or deployed within the Mesosphere cluster. Your application is packaged and deployed consistently in either case. This makes ConductR's standalone mode ideal for provisioning smaller testing and development clusters. Each team can quickly and easily be provisioned with its own sandbox cluster, enabling it to safely perform full integration testing prior to staging in the enterprise cluster.

One feature to be certain to look for your deployment solution is dynamic ingress proxying. The vast majority of ingress traffic of most deployments is over ports 80 and 443. Within the cluster, bundle executions must be bound to dynamically assigned ports in order to avoid collisions. The cluster must provide a dynamic proxy solution so that you can easily ingress to public endpoints. If not, operators must provision means to update proxies or IP addresses in DNS.

One of the most common requirements from a control plane is the ability to perform rolling updates of services. This dovetails with the separation of application and configuration, or the ability to modify configuration distinctly from and without modification of development artifacts. When updating application versions, you want to roll the new versions in, migrating load to the updated services, and then terminating the old instances.

Containers are an inherent part of microservices. The Open Containers Initiative (OCI) was established by Docker and others to maintain open specifications for container images and runtimes. The rkt engine, for example, is an implementation of the OCI *app container specification*. The OCI develops and maintains runC, the container runtime started and donated by Docker and still used as the core of Docker engine. Use OCI to avoid being locked to a particular vendor or workflow while retaining the benefit of being battled-tested in production. ConductR directly supports the OCI image-spec format, enabling you to utilize container technologies without committing to a long-term relationship with any one vendor. For composability, ConductR's `bndl` tool provides for connecting, or *piping*, `docker save` into the cluster's `load` command. This enables rapid development cycles without tightly binding your development workflow to the Lightbend solution. Docker and other image-spec-compliant images are executed directly in runC.

NOTE

Here's an example of realizing resilience by isolation. Instead of pushing a Dockerfile into the deployment tools, you load the full image from `docker save`. This avoids the container engine needing to fetch layers from a registry before it can scale a service. Fetching increases the time required to start executing, while introducing the chance of failure if all layers cannot be fetched.

Akka clustering and other masterless, gossip-based technologies, peer-node applications, and data engines can be a challenge for some deployment environments. Peer application instances must be able to discover and communicate with their peers. Schedulers may make no consideration of application cluster formation, launching all instances in parallel and making seed node determination more complicated. Ensure your cluster scheduler provides such features whenever using applications that require them. Finally, compatibility between peer systems should be part of the deployment in order to enable rolling upgrades across incompatible revisions without further complicating the migration. All applications using the Akka clustering feature, including Lagom and Play applications using clustering via the default Akka system, need to include clustering and seeding requirements in their deployment plans. ConductR is cluster-aware and fully supports seeding for applications using Akka clustering.

Application-Centric Logging, Telemetry, and Monitoring

Without application-specific logging, telemetry, and monitoring, you are flying blind. We also know that meaningful, actionable data is far more valuable than petabytes of raw telemetry. How many messages are in a given queue and how long it is taking to service those events is far more indicative of the service response times that are tied to your service-level agreements. Disk, CPU, GPU, and network numbers are useful in that they can help us detect problems and bottlenecks, but are just too coarse-grained to be useful toward understanding the cause. With multiple services running on a node and the use of constraints such as kernel cgroups (control groups), node resource utilization is but one part of the picture. Return codes and rates from the proxying layer are a more reliable source for overall health than most host metrics. Look to the number of Actors, their mailbox size, the number of messages processed, and queue sizes will look to when you need to better understand the status of your service.

Good, useful metrics, events, and log messages come from the application. There is simply no better source for this data than the source code of the application itself. The messages, events, and statistics are designed, developed, tested, revised, and re-hardened by the teams as they work on and use the service. Most often, you are primarily

interested in the logs of all instances of a single service. You don't particularly need to know which nodes the service is executing on. So long as those nodes are healthy and providing resources as expected, the location of the node only becomes a concern with regards to availability zone, regional-level distribution, and as part of resilience planning. Furthermore, you often do not know which instance of a service serviced a given request, produced the error messages, or was otherwise of interest. You often know which service to look at first, but the client rarely knows exactly which instance in the cluster serviced its request. After clicking into the service of interest from the dashboard, you need meaningful information, and most of that comes from the application-emitted data.

Here, too, the newer generations of systems are using the best practices of application development. Log messages, bundle events, utilization metrics, and telemetry should be streamed using messaging, with publish and subscribe semantics enabling consumption by services such as auto-scaling and alerting. Log messages are streamed using the Syslog protocol for compatibility with existing tools, as well as most services from AppDynamics to DataDog.

Visual dashboards are literally the face of your cluster. The dashboard must be truly be indicative of system status in order to provide confidence. Dashboards should be easy to assemble, self-discovering much of the infrastructure. Like testing, if the assembly is too difficult, it may not happen. When you need additional information about services, the dashboards should graphically connect users to the service log, events, and telemetry. Command-line tools can be composed to build very useful scripts, but casual users will generally prefer a discoverable graphical interface.

Application-Centric Process Monitoring

A fundamental aspect of monitoring is that the supervisory system automatically restarts services if they terminate unexpectedly. A nonzero exit code from a process is a good indicator that it didn't expect to terminate. The scheduler, as long as the cluster has resources available, will have the desired number of instances of all service bundles running. If there are not enough, more instances will be started somewhere in the cluster. If there are too many instances, some will be shut down. This is a basic function of the scheduler.

Preventing a split-brain cluster, however, is far from basic. Network partitions are a reality of distributed computing. You must have automatic split-brain resolution features that both quarantine orphaned members and signal affected applications. Agents being monitored by downed schedulers should seek alternative members or down the node if unable to connect. Once connectivity is restored, the system should self-heal.

Telemetry can produce vast amounts of data, so you need quality, not quantity. Too much telemetry will congest networks and constrain resources. Effective monitoring requires an events model from which services can subscribe to events from both the services and their orchestration layer in order to make intelligent decisions or take corrective actions. This enables the application services to make key metrics and events available to all interested services.

Elastic and Scalable

Elastic scaling is one of the most requested features of cloud deployments. What you need is effective scaling.

The first step in being scalable is application design. Without the isolation and autonomy previously discussed, an application cannot be scaled simply by adding additional nodes to the cluster. Stateful and clustered applications have additional considerations, such as local shard replication when moving nodes of data stores.

There are two aspects to scaling: scaling the number of instances of a service and scaling the resources of a cluster. Clusters need some amount of spare capacity or headroom. For example, if a node should fail, you will generally want to leave enough headroom to restart the affected services elsewhere without having to provision. When existing resources cannot provide all the desired instances, additional nodes must be provisioned.

Autoscaling is the scaling of instances and/or nodes, up or down as needed, automatically. Microservices come in systems, and changes to one service impact other inhabitants of the system. Consider a check out queue in which you do not want customers waiting for a long time to check out. Increasing the number of cashiers does not help if they are not the bottleneck. If the cashiers are waiting for the sales terminal service, adding more cashiers would only increase load on the already overloaded terminal service. In autoscaling, it is also easy to create distributed thundering herd problems.

The need for autonomy and isolation for resilience applies to all aspects of the deployment of Reactive microservices. When scaling instances, you need the entire container image, dependencies and all. Dependency resolution must be a build-time concern if a container engine is to be certain of its ability to run an image. Even if you can assure that all required objects will be available, they still need to be fetched from the repositories, and the scale operation cannot complete in isolation from the registries. ConductR's bundles contain the full `docker save` archive to avoid fetching layers from Docker repositories when running the bundled service. This results in services being able to start quickly and reliably.

Likewise, when provisioning nodes, you need a full node image to provision with. Upon launch, the new node may need an IP address or two to help it join the cluster. You do not want to be dependent upon the completion of cookbooks and playbooks when nodes are needed. If some nodes have additional role-specific configurations, such as the installation and configuration of a proxy such as HAProxy, for public nodes, that should be included in the public node provisioning image. Infrastructure failures can exacerbate the situation as other users load the system in efforts to minimize damage. You simply do not want the risk of being dependent on external resources in such situations. Teams looking to squeeze every bit of fault tolerance out of their cluster may extend this isolationism into the node instance itself, avoiding Amazon Elastic Block Store (EBS)-backed instances, for example.

Now that we've examined the theoretical benefits of Reactive on a deployment system, let's try it out hands on! In the next chapter you'll deploy Reactively using Lightbend Enterprise Suite. You will have the opportunity to try various failure scenarios and observe self-healing in action, firsthand.

CHAPTER 3
Deploying Reactively

> The credit belongs to the man who is actually in the arena.
>
> —Theodore Roosevelt, *Citizenship in a Republic*

Microservices require isolation for elasticity and resilience. Yet by themselves, they are not very useful. Microservices are deployed and managed as systems on clusters. If your deployment system provisions you with well-designed, composable interfaces for handling all the interstitial details, you can deliver your Reactive applications without coding the application to be tightly coupled with its delivery and deployment systems. In this chapter you will deploy a Lagom microservices application using Lightbend Enterprise Suite. You will be using ConductR, which utilizes Akka Actors.

The source code for the example application that you will deploy is available on GitHub. The project's *deploy.md* file contains additional information and updates about deploying and testing the sample application project in a wider variety of environments using the latest versions.

The Lagom Chirper application is an example of a service similar to that of Twitter. Instead of *tweets,* Chirper users post *chirps.* To be clear, this is only an example and not a complete application. If you are new to developing Reactive microservices, the project can help you better understand how to build a Reactive microservices system using Lagom. You will be using the Chirper application for experimenting with the self-healing properties of Lightbend Enterprise Suite.

The Reactive Service Orchestration feature of Lightbend Enterprise Suite is provided by a project called ConductR, which I discussed in Chapter 2. ConductR also provides a Developer Sandbox for deploying services into a local production-like environment. The sandbox is lightweight and simple to run, making it ideal for developer and CI validation testing.

Multinode clusters can be created in DC/OS using Universe, or in Amazon Web Services using Cloudformation and Ansible playbooks. Visit the getting started guide for additional details and information.

This chapter assumes that you're using the Lightbend Enterprise Suite Developer Sandbox. If you are using a multinode cluster, you can alternatively use a Windows, macOS, or Linux host to execute the `conduct` CLI commands on a remote cluster, such as in the cloud. If you are able to SSH to one of the cluster nodes, the CLI is installed on all nodes by default. If you are attempting to access an existing cluster, such as one installed on DC/OS, you may need to run the CLI from a system with specific access to the cluster, such as by VPN or SSH tunneling. In such cases you may need to contact your IT department for details. Follow the instructions for installing the CLI in the ConductR documentation to install that latest CLI release.

The `visualizer` is an application that demonstrates the use of Server Sent Events (SSEs) to visualize where services are executing in the cluster. If the `visualizer` bundle is not available on your cluster, use `conduct run visualizer` and then `conduct load visualizer` to start it.

Getting Started

The exercises in this chapter utilize Java applications. Please ensure you have a recent Java 8 JRE available on the system. Both OpenJDK and Oracle's Java are widely used. You will need git, and Maven or sbt installed on your system in order to build the application. You will need to be able execute `docker` commands in order to run the examples I present, as they utilize Docker images. ConductR's command-line tools include resolvers for the DockerHub and Bintray registries. This makes it simple to deploy public and private

images from these registries. For example, to load the latest `nginx` image with the default configuration from DockerHub in a cluster, use:

```
conduct load nginx
```

The examples in this chapter use the HAProxy image from DockerHub to enable the dynamic proxying feature. ConductR's dynamic proxy feature routes advertised endpoints to worker instances for public ingress. The proxy configuration is dynamically updated as container state changes in the cluster. All proxying is done by HAProxy. The Lightbend-provided `conductr-haproxy` bundle subscribes to bundle events and dynamically updates the HAProxy configuration in response. This ensures HAProxy is always current, Reactively. For more information about loading Docker and other OCI container images into ConductR, see the project documentation.

I recommend registering for the free deployment license for running multiple instances of your services at Lightbend.com. This license allows for the use of up to three agent nodes in production, allowing for the scaling of up to three instances of each service.

Please note that at least three nodes are required to use the network partition, or Split-Brain Resolution (SBR) feature, of Lightbend Enterprise Suite. It is not possible to form an initial quorum with less than three nodes. Given the near certainty of numerous network partitions during the deployment of a distributed application, if you are delivering clustered production applications to the cloud, then you'll want to be certain to enable this feature.

Developer Sandbox Setup

The Developer Sandbox is a lightweight cluster for testing on a developer's workstation or from within the integration tests. Because all of the cluster's schedulers and agents run on a single host, it is *not* suitable for production deployments. The Lightbend Enterprise Suite Developer Sandbox currently supports macOS and Linux x64. If you are using Windows or other operating systems, please note that the use of a Linux VM, such as the current major release of Ubuntu, is suggested. The Lightbend Enterprise Suite getting started guide has detailed information and more options for running the sandbox.

Follow the getting started guide to install and configure Lightbend Enterprise Suite on your system. You will use the `sandbox` and `con duct` commands from the command-line interface (CLI) to control the cluster. The examples in this report use version `2.1.0`. See the getting started guide for the latest version information.

Clone the Example

To get started with the example application, clone the Lagom Chirper example into your machine:

```
mkdir -p ~/examples
cd examples
git clone git@github.com:lagom/activator-lagom-java-chirper.git
```

Deploying Lagom Chirper

Now you're ready to deploy Lagom Chirper. To get started, launch the sandbox:

```
sandbox run 2.1.0 -n 3 -f visualization
```

Next, go to the cloned Lagom Chirper directory:

```
cd ~/examples/activator-lagom-java-chirper
```

Then deploy the Lagom Chirper application. It is packaged as both SBT and Maven. Use whichever build tool you're most comfortable with.

Maven:

```
./mvn-install
```

SBT:

```
sbt install
```

When you deploy Chirper for the first time, it may take a few minutes since the build tool will need to download all of the project's dependencies. On subsequent deploys, these tasks will complete significantly faster as the build tool does not need to re-fetch these files.

Once the artifacts are downloaded, the following tasks will be performed by the build tool:

- Build the bundle for the Activity Stream, Chirp, Friend, and Front-End services, respectively.

- Deploy and run Cassandra using the generated local configuration bundle.
- Deploy and run the Activity Stream, Chirp, Friend, and Front-End bundles, respectively.

NOTE

A bundle is an archive of components along with metadata (a bundle descriptor file) describing how the files should be executed. Bundles are similar to executable JARs, with the primary difference being that they are not constrained to executing just JVM code. Bundles are named using a digest of their contents so that their integrity can always be assured. Bundles represent a unit of software that may have a release cycle which is distinct from other bundles.

Once deployed, use the `conduct info` command to inspect the state of the deployed services. When you run `conduct info`, you should see something similar to this:

```
$ conduct info
ID              NAME                  VER  #REP  #STR  #RUN
89fe6ec         activity-stream-impl   v1     1     0     1
73595ec         visualizer             v2     1     0     1
bdfa43d-e5f3504 conductr-haproxy       v2     1     0     1
6ac8c39         load-test-impl         v1     1     0     1
9a2acf1-44e4d55 front-end              v1     1     0     1
3349b6b         eslite                 v1     1     0     1
01dd0af         friend-impl            v1     1     0     1
d842342         chirp-impl             v1     1     0     1
1acac1d         cassandra              v3     1     0     1
```

NOTE

To keep this report reasonably formatted, I've abbreviated many console outputs. I've aimed to retain the important details, but you should not expect your console output to match those shown in this report.

The frontend of the Chirper application will be available as well. Open *http://192.168.10.1:9000/* in your browser to visit the Chirper frontend. Congratulations! You have successfully deployed Lagom Chirper into the Developer Sandbox!

Lagom Chirper is an example application that showcases Lagom features with functionality similar to Twitter. Lagom Chirper comprises of the following services:

`Chirp` *service*
: Responsible for the storage of `chirps`, and providing the interface to obtain the stored `chirps`.

`Friend` *service*
: Responsible for the storage of `users`, and managing the relationship between the stored `users` and their `friends`.

`Activity Stream` *service*
: Provides a stream of chirps given a particular user. Dependent on the `Chirp` service and `Friend` service.

`Front-End` *service*
: Provides the web-based user interface.

The `Activity Stream`, `Chirp`, and `Friend` services are written in Lagom, while the `Front-End` service is written in Playframework.

Reactive Service Orchestration

When an application is deployed, ConductR will ensure every instance is started sequentially. This enables applications to form peer clusters in a consistent and reliable manner. Once an instance has been started successfully and confirmed to be healthy, only then will the attempt to start the next instance proceed. An application signals to the cluster that it's ready to start by hitting a REST endpoint, or alternatively by invoking a check command that will make the call to the same REST endpoint. If your application is written in Lagom or Play, the health check will be invoked automatically by the inclusion of `conductr-lib`. For non-Lagom and non-Play applications, the signaling application state page in the ConductR documentation has further details. If an application fails to start, ConductR will attempt to restart the application for a given number of configurable attempts. This predictable application start behavior allows scripting of the deployment using simple bash script and the CLI. This simplicity is an advantage as there is no need for operations teams to learn a new language or DSL in order to start creating their own deployment script.

View the deployment script from Lagom Chirper to see an example of one. Use the command for your respective build tool.

Maven:

```
cat ./mvn-install
```

SBT:

```
sbt generateInstallationScript && cat target/install.sh
```

For applications written in Lagom with Akka Clustering enabled, ConductR is able to form the Akka cluster automatically. For non-Lagom applications, the Akka Clustering page in the documentation provides the instructions so that ConductR can form the Akka cluster for your application.

You can use the Reactive Service Orchestration and Service Discovery features of Lightbend Enterprise Suite to create peer-node clusters for other services and frameworks as well. The ConductR-Cassandra bundle is a project in which ConductR orchestrates the formation of a Cassandra cluster. Lightbend recommends OCI bundling for all non-JVM services. The Postgres BDR project demonstrates how to form a Postgres cluster using a Docker bundled service.

Elasticity and Scalability

It's simple to scale services up and down. Let's scale `Activity Stream` service up to `2` instances by executing the following command:

```
conduct run activity-stream-impl --scale 2
```

Output similar to the following will be displayed. The CLI tool will wait for `Activity Stream` to be scaled to `2` instances:

```
$ conduct run activity-stream-impl --scale 2
Bundle run request sent.
Bundle 39f36b39adcd108 waiting to reach expected scale 2
Bundle 39f36b39adcd108 has scale 1, expected 2
Bundle 39f36b39adcd108 expected scale 2 is met
Stop bundle with: conduct stop 39f36b3
Print ConductR info with: conduct info
```

When services are scaled up, the declared resource profile will be used to determine where the service instance will run. An agent node that has the requested resources available, as defined in units of CPU, memory, and disk space, and is not already executing the same bundle will be selected to start a new instance. The resource profile is declared as part of a bundle configuration. Bundle configuration includes `roles` for grouping workload with resources. When

role matching is enabled, only agent nodes that offer all the `roles` declared by the `roles` settings will be eligible to run the service.

To bring `Activity Stream` service back to `1` instance, execute the following command:

```
conduct run activity-stream-impl --scale 1
```

Output similar to the following will be displayed:

```
$ conduct run activity-stream-impl --scale 1
Bundle run request sent.
Bundle 39f36b39adcd108 waiting to reach expected scale 1
Bundle 39f36b39adcd108 expected scale 1 is met
Stop bundle with: conduct stop 39f36b3
Print ConductR info with: conduct info
```

Process Resilience

ConductR monitors the service processes that it has launched and ensures that the requested number of instances is being satisfied. When a service process terminates unexpectedly, new instances of the bundle will be started, seeking to meet the requested scale. If conditions require multiple instances of a service to be started, ConductR will ensure the processes are launched sequentially, one at a time, avoiding unnecessary and often disruptive application cluster state changes. In the situation where a service instance terminates due to loss of a node, ConductR will attempt to start the service on one of the remaining machines where the service is not already running until the requested scale is met.

In the node failure scenario, it's possible that the number of requested instances cannot be met due to an insufficient amount of resources being available. In such cases, when a replacement node is commissioned and joins the cluster, ConductR will automatically attempt to start the interrupted service until the number of requested instances is met again. If a service is loaded with a configuration bundle, that bundle configuration will always be applied to new instances of the service by the same bundle identifier.

This resilient behavior relieves operations from the burden of having to restart the services whenever there's an unexpected termination. Should the service interruption be caused by hardware failure, operations can focus on node provisioning and let the cluster handle the recovery of the service instances.

Let's simulate the automatic process recovery by terminating one of the services within Lagom Chirper. Scale `Activity Stream` to `2` instances by running the following command:

```
conduct run activity-stream-impl --scale 2
```

Once scaled, the overall state should be similar to the following. The `activity-stream-impl`, which is the bundle name of the `Activity Stream` service, now has `2` running instances:

```
$ conduct info
ID               NAME                  VER  #REP  #STR  #RUN
89fe6ec          activity-stream-impl   v1     1     0     2
73595ec          visualizer             v2     1     0     1
bdfa43d-e5f3504  conductr-haproxy       v2     1     0     1
6ac8c39          load-test-impl         v1     1     0     1
9a2acf1-44e4d55  front-end              v1     1     0     1
3349b6b          eslite                 v1     1     0     1
01dd0af          friend-impl            v1     1     0     1
d842342          chirp-impl             v1     1     0     1
1acac1d          cassandra              v3     1     0     1
```

You should be able to see `2` process IDs belonging to `Activity Stream` service. Take note of the value of both process IDs:

```
pgrep -f activity-stream-impl
```

Let's kill one of the processes:

```
pgrep -f activity-stream-impl | head -n 1 | xargs kill
```

Eventually, you should see `2` process IDs belonging to `Activity Stream` service, with a new process ID replacing the one you killed. The new process will start quite quickly, and it's very likely you will see the new process ID when you run this command:

```
pgrep -f activity-stream-impl
```

You can scale `Activity Stream` back down to `1` instance if you wish:

```
conduct run activity-stream-impl --scale 1
```

Rolling Upgrade

Next, you will perform a rolling upgrade of the `Friend` service. Rolling upgrades on ConductR are relatively straightforward. The new version has a new bundle identifier, making the new and old versions distinct. A new version of a service can be deployed and run alongside an existing version. Should the new version expose the same endpoint as the existing version, while running alongside each

other the traffic from the proxy will be delivered to both new and existing versions in round-robin fashion.

There is an instance of `Friend` service already running—look for the bundle with `friend-impl` as its `NAME`. Note the `ID` of the `friend-impl` service—it has `01dd0af` as its value. You call this identifier "Bundle ID."

NOTE

The bundle IDs shown here will not be the same as your bundles, even for the same services.

```
$ conduct info
ID               NAME                  VER  #REP  #STR  #RUN
89fe6ec          activity-stream-impl   v1     1     0     1
73595ec          visualizer             v2     1     0     1
bdfa43d-e5f3504  conductr-haproxy       v2     1     0     1
6ac8c39          load-test-impl         v1     1     0     1
9a2acf1-44e4d55  front-end              v1     1     0     1
3349b6b          eslite                 v1     1     0     1
01dd0af          friend-impl            v1     1     0     1
d842342          chirp-impl             v1     1     0     1
1acac1d          cassandra              v3     1     0     1
```

First, build a new version of the `Friend` service. Even if you don't make code changes, a clean build will produce a new bundle which you can use to perform a rolling upgrade.

Maven:

```
mvn clean package docker:build && \
docker save chirper/friend-impl | bndl --no-default-check \
--endpoint friend --endpoint akka-remote | conduct load
```

SBT:

```
sbt friend-impl/clean friend-impl/bundle:dist && \
conduct load -q $(find friend-impl/target -iname \
"friend-impl-*.zip" | head -n 1) | xargs conduct run
```

You now have a new instance of the `Friend` service running alongside the original one. The original `Friend` service has `01dd0af` as its bundle ID, while the new one has `87375e6`.

```
$ conduct info
ID               NAME                  VER  #REP  #STR  #RUN
89fe6ec          activity-stream-impl   v1     1     0     1
73595ec          visualizer             v2     1     0     1
bdfa43d-e5f3504  conductr-haproxy       v2     1     0     1
```

```
6ac8c39          load-test-impl        v1     1     0     1
9a2acf1-44e4d55  front-end             v1     1     0     1
3349b6b          eslite                v1     1     0     1
01dd0af          friend-impl           v1     1     0     1
87375e6          friend-impl           v1     1     0     1
d842342          chirp-impl            v1     1     0     1
1acac1d          cassandra             v3     1     0     1
```

At this point, any HTTP requests made to the `Friend` service through the proxy will be delivered in a round-robin fashion between `01dd0af` and `87375e6`. The lookup to the `Friend` service through the ConductR service locator will also be distributed between `01dd0af` and `87375e6`.

Stop and undeploy the original `Friend` service. (Note that you use the bundle ID `3ff82bf` to refer to the original `Friend` service, as the name `friend-impl` is associated to both `01dd0af` and `87375e6`.)

```
conduct stop 01dd0af
conduct unload 01dd0af
```

You have now performed a rolling upgrade of the `Friend` service!

Dynamic Proxying

Lightbend Enterprise Suite's Dynamic Proxying feature provides location transparency of your services to their clients. The caller of your services only needs to know the address of the proxy in order to access the services. Service location information is updated Reactively, absolving callers from needing to be aware of changes in service instance locations, such as those due to scaling changes or rolling upgrades.

ConductR allows services to expose their endpoints to be accessible via the proxy. The proxies are used for ingress traffic to the application services. The proxy configuration will be updated as services are scaled up or down, ensuring that the requests being made to these services via the proxy will be routed to an available instance.

Let's scale the Lagom Chirper `Activity Stream` up and down, and observe the automatic changes made to the proxy configuration. In the Developer Sandbox, HAProxy is started as a Docker image. To view the HAProxy configuration, execute the following command:

```
docker exec -ti sandbox-haproxy \
cat /usr/local/etc/haproxy/haproxy.cfg
```

You should see a HAProxy `frontend` configuration similar to the following:

```
frontend default-http-frontend
 mode http
 bind 0.0.0.0:9000
 acl 1095435chirpsvc-acl-0 path_beg /api/chirps/live
 use_backend 109543-chirpsvc-backend-0 if 109543-chirpsvc-acl-0
 acl 109543-chirpsvc-acl-1 path_beg /api/chirps/history
 use_backend 109543-chirpsvc-back-1 if 109543-chirpsvc-acl-1
 acl f13f7a-activitysvc-acl-0 path_beg /api/activity
 use_backend f13f7a-activsvc-back-0 if f13f7a-activitysvc-acl-0
 acl 29006c-friendsvc-acl-0 path_beg /api/users
 use_backend 29006c-friendsvc-back-0 if 29006c-friendsvc-acl-0
 acl 64c310-44e4d5-web-acl-0 path_beg /
 use_backend 64c310-44e4d5-web-back-0 if 64c310-44e4d5-web-acl-0
```

NOTE

For those unfamiliar with HAProxy, the HAProxy `frontend` accepts the incoming request, then routes the request to the corresponding HAProxy `backend`. The HAProxy `backend` has one or more addresses that are able to service the request, and by default HAProxy will route the request to each address in round-robin fashion.

The Lagom Chirper `Activity Stream` exposes an endpoint on the `/api/activity` path. This endpoint is called `activityservice`. In the HAProxy `frontend` configuration shown above, requests matching the `acl` that has the name ending with `activityservice-acl-0` will be routed to the HAProxy `backend` configuration that has the name ending with `activityservice-backend-0`. The HAProxy `backend` configuration that has the name ending with `activityservice-backend-0` should be similar to the following:

```
backend f13f7a-activityservice-backend-0
  mode http

  server 19216810310452 192.168.10.3:10452 maxconn 1024
```

Since there is only `1` instance of the Lagom Chirper `Activity Stream` running, there's only one single address listed. Execute the following command to scale the Lagom Chirper `Activity Stream` to 2 instances:

```
conduct run activity-stream-impl --scale 2
```

View the HAProxy configuration once more by executing the following command:

```
docker exec -ti sandbox-haproxy \
cat /usr/local/etc/haproxy/haproxy.cfg
```

The `backend` configuration will now have an additional address automatically added. This newly added address points to the newly started Lagom Chirper `Activity Stream` service instance. If you access the `Activity Stream` URL on `http://192.168.10.1:9000/api/activity` repeatedly, HAProxy will round-robin between these two addresses.

```
backend f13f7a-activityservice-backend-0
  mode http

  server 19216810310452 192.168.10.3:10452 maxconn 1024
  server 19216810210822 192.168.10.2:10822 maxconn 1024
```

Similarly, the configuration will be updated when the Lagom Chirper `Activity Stream` service is scaled down. When the Lagom Chirper `Activity Stream` service is stopped, the entry from the HAProxy configuration is removed.

ConductR allows for the customization of the HAProxy configuration by providing HAProxy with a configuration template file. This enables operators to exercise full control over how proxying is handled, while still having the configuration updated automatically as executions change. Refer to the Dynamic Proxy Configuration documentation for more details.

Service Locator

Lightbend Enterprise Suite's Service Discovery feature allows a caller of a service to lookup the service's address, thus allowing the request to be made to the correct address. This pattern is particularly important in the deployment of microservice-based applications, as services are expected to be scaled up, down, and relocated for various reasons. The list of address for all services running within the system needs to be maintained and kept current. When using an orchestration product that comes with a built-in service registry such as ConductR, our services can utilize the Service Discovery feature without adding the complexity of additional daemons to our cluster.

Let's test the Service Discovery feature by attempting to look up a particular user from the `Friend` service within the Lagom Chirper example. First, you need to register a user. Visit the `Front-End`

address at *http://192.168.10.1:9000/*, and click on the "Sign Up" button to view the registration page. Enter `joe` for both the `Username` and `Name`, and click the Submit button. The user `joe` can now be looked up from the `Friend` service through the proxy URL:

```
curl http://192.168.10.1:9000/api/users/joe
```

You should see the following JSON response:

```
{"userId":"joe","name":"Joe","friends":[]}
```

Next, try to look up `joe` from the `Friend` service through the service locator instead. To do that, you will need to look up the `Friend API` from the service locator. Issue the following command to see the endpoints that can be looked up via the service locator:

```
conduct service-names
```

You should see output similar to the following:

```
$ conduct service-names
SERVICE NAME     BUNDLE ID        BUNDLE NAME           STATUS
activityservice  89fe6ec          activity-stream-impl  Running
cas_native       1acac1d          cassandra             Running
chirpservice     d842342          chirp-impl            Running
elastic-search   3349b6b          eslite                Running
friendservice    01dd0af          friend-impl           Running
loadtestservice  6ac8c39          load-test-impl        Running
visualizer       73595ec          visualizer            Running
web              9a2acf1-44e4d55  front-end             Running
```

The `BUNDLE NAME` for the `Friend` service is called `friend-impl`, and it exposes its endpoint called `friendservice`. ConductR exposes its service locator on port `9008`, and in the Developer Sandbox the service locator is accessible on *http://192.168.10.1:9008*.

Find the addresses for `friendservice` by executing the following command:

```
curl -v http://192.168.10.1:9008/service-hosts/friendservice
```

You should see a JSON list containing the host address and bind port of the `friendservice`, similar to the following:

```
["192.168.10.2:10785"]
```

Let's scale the `Friend` service to `2` instances:

```
conduct run friend-impl --scale 2
```

You should see the list of `friendservice` host addresses updated accordingly:

```
$ curl http://192.168.10.1:9008/service-hosts/friendservice
["192.168.10.2:10785","192.168.10.3:10373"]
```

ConductR monitors the services it has started and automatically updates the CRDT (conflict-free replicated data type) of service locations whenever there is a change in endpoint locations, regardless of the cause. Since the service address list is automatically maintained by ConductR, applications are relieved from the burden of registering and deregistering themselves with the service registry. ConductR's service locator also provides an HTTP redirection service to the `friendservice` endpoint.

Execute the following command to invoke the `friendservice` endpoint via HTTP redirection:

```
curl http://192.168.10.1:9008/services/friendservice/api/users/joe \
-L
```

You should see the following JSON response:

```
{"userId":"joe","name":"Joe","friends":[]}
```

Let's examine the HTTP request in detail. The URL of the request is *http://192.168.10.1:9008/services/friendservice/api/users/joe*, and it is comprised of the following parts. The *http://192.168.10.1:9008/services* is the base URL of the HTTP redirection service provided by the service locator. The next part of the URL is *friendservice*, which is the name of the endpoint you would like to be redirected to. The remaining part of the URL, */api/users/joe*, forms the actual redirect URL to the `friendservice` endpoint. You can view the request and response by executing the `curl` command and passing the verbose switch, `-v`:

```
curl http://192.168.10.1:9008/services/friendservice/api/users/joe \
-v -L
```

You should see the following output:

```
$ curl http://192.168.10.1:9008/services/friendservice/api/users/joe \
-v -L
*   Trying 192.168.10.1...
* Connected to 192.168.10.1 (192.168.10.1) port 9008 (#0)
> GET /services/friendservice/api/users/joe HTTP/1.1
> Host: 192.168.10.1:9008
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 307 Temporary Redirect
< Location: http://192.168.10.2:10785/api/users/joe
< Cache-Control: private="Location", max-age=60
```

```
< Server: akka-http/10.0.0
< Date: Tue, 13 June 2017 06:17:37 GMT
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 50
<
* Ignoring the response-body
* Connection #0 to host 192.168.10.1 left intact
* Issue another request to this URL:
*   'http://192.168.10.2:10785/api/users/joe'
*   Trying 192.168.10.2...
* Connected to 192.168.10.2 (192.168.10.2) port 10785 (#1)
> GET /api/users/joe HTTP/1.1
> Host: 192.168.10.2:10785
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 42
< Content-Type: application/json; charset=utf-8
< Date: Tue, 13 June 2017 06:17:37 GMT
<
* Connection #1 to host 192.168.10.2 left intact
{"userId":"joe","name":"Joe","friends":[]}
```

Note there are two HTTP request/response exchanges in the output above. The first response is replied with HTTP status code `307`, which is a redirect to the address where one of the `friendservice` endpoints resides. The redirect location is declared by the Location response header at *http://192.168.10.2:10785/api/users/joe*.

The `curl` command is set to automatically follow redirect by supplying the `-L` flag. As such, the second HTTP request is then automatically made to *http://192.168.10.2:10785/api/users/joe*. The service locator HTTP redirection feature allows performing service lookup with minimal change to the caller's code. The caller code does not need to bear the burden of performing address lookup prior to the endpoint call. Instead, the service locator will perform the address lookup internally on the caller's behalf, resulting in the HTTP redirection to the correct address for the caller to follow.

Note that HTTP `307` works with other HTTP verbs too, so the redirect works with HTTP Post with JSON payload or form parameters, for example. From the developer's perspective, this would mean the HTTP request's relative path and payload to the endpoint remain constant, only the base URI where the endpoint resides will be different.

From the application's perspective, the Service Locator Base URL will be provided by the `SERVICE_LOCATOR` environment variable when running within ConductR. When the `SERVICE_LOCATOR` environment variable is present, it will configure the base URL of the endpoint by appending the endpoint name to the `SERVICE_LOCATOR`. The HTTP request made to the base URL configured in this manner will be automatically redirected to the desired endpoint.

If the `SERVICE_LOCATOR` environment is not present, the base URL of the endpoint can fall back to a default value. This is useful for running the caller in a development environment, for example.

Consolidated Logging

Reviewing application log files is part of regular support activities. With applications built using microservices, the number of log files to be inspected can grow significantly. The effort to inspect and trace these log files grows tremendously when each log file is located on separate machines.

ConductR provides an out-of-the-box solution to collect and consolidate the logs generated by the application, deployed and launched through ConductR itself. Once consolidated, the logs then can be viewed using the `conduct logs` command. Let's view the log from the visualizer bundle by running the following command:

```
conduct logs visualizer
```

You should see the log entries from the visualizer application, similar to the following:

```
$ conduct logs visualizer
TIME      HOST  LOG
14:05:34  les1  [info] play.api.Play - Application started (Prod)
14:05:34  les1  [info] application - Signalled start to ConductR
14:05:34  les1  [info] Listening for HTTP on /192.168.10.2:10609
```

The `Listening for HTTP on 192.168.10.2:10609` entry indicates the visualizer application is started and bound to the `192.168.10.2` address. Since `192.168.10.2` is the address alias that points to your local machine, that `HOST` column will always be populated by the host address of your local machine. In this example, `les1` is the name of the local machine that visualizer is running on.

To see ConductR's consolidated logging feature in action, scale the visualizer to 3 instances:

```
conduct run visualizer --scale 3
```

Once scaled to 3 instances, you can view the logs consolidated from all the visualizer instances by executing the following command:

```
conduct logs visualizer
```

You should see something similar to the output below. The log entries are consolidated from all three instances of visualizer running on 192.168.10.1, 192.168.10.2, and 192.168.10.3:

```
$ conduct logs visualizer
TIME      HOST  LOG
14:05:34  les1  [info] application - Signalled start to ConductR
14:05:34  les1  [info] Listening for HTTP on /192.168.10.2:10609
14:16:31  les1  [info] play.api.Play - Application started (Prod)
14:16:31  les1  [info] application - Signalled start to ConductR
14:16:31  les1  [info] Listening for HTTP on /192.168.10.3:10166
14:16:35  les1  [info] play.api.Play - Application started (Prod)
14:16:35  les1  [info] application - Signalled start to ConductR
14:16:35  les1  [info] Listening for HTTP on /192.168.10.1:10822
```

The logs collected by ConductR are structured according to Syslog's definition of structured data. This structure is discussed in the Logging Structure page. The collected log entries can be emitted to either Elasticsearch or RSYSLOG.

When Elasticsearch is enabled, the log entries are indexed and become searchable. The Kibana UI can be installed to provide a user interface for querying these log entries. ConductR consolidated logging works with both an existing Elasticsearch cluster outside of Lightbend Enterprise Suite, or the Elasticsearch cluster managed by Lightbend Enterprise Suite.

By default, the Developer Sandbox starts up with a severely slimmed-down version of Elasticsearch called "eslite" to be used for development purposes only. Alternately, to enable the actual Elasticsearch on the Developer Sandbox, provide the `-f` logging option when executing `sandbox run`. The `-f logging` option will also enable the Docker-based Kibana UI, accessible through *http://192.168.10.1:5601*.

If you wish to see the actual Elasticsearch bundle in action, execute the following command.

Maven:

```
sandbox run 2.1.0 -n 3 -f visualization -f logging
./mvn-install
```

SBT:

```
sandbox run 2.1.0 -n 3 -f visualization -f logging
sbt install
```

The production Elasticsearch instance is configured with the JVM heap sized to 1 GB. So be certain that your machine has sufficient memory resources to run Elasticsearch with all Lagom Chirper services.

When using RSYSLOG, apart from directing the logs into the RSYSLOG logging service, the logs can be sent to any log aggregator that speaks the syslog protocol, such as Humio.

Network Partition Resilience

Due to its distributed nature, network partitioning is one of many failure scenarios that microservice-based applications must contend with. A network partition occurs when parts of the network are intermittently reachable, or unreachable due to network connectivity issues. The possibility of a network partition occurring is quite real, particularly when deploying to public cloud infrastructure where there is limited control of the underlying network infrastructure.

Often for both resiliency and performance, multiple instances of a service are started and their states are synchronized by clustering the instances together. When a network partition occurs, one or more instances of these services can become separated from the other instances. Therefore, updates to the cluster state may not reach the orphaned instances, which can lead to inconsistencies or corruption of the data being managed by those instances.

ConductR's out-of-the-box defense against network partitions, as previously noted, is the SBR feature. This feature is automatically enabled for clusters of three or more nodes. You can also use the SBR feature as your downing strategy in the development of your Akka-based Reactive applications. ConductR is comprised of core nodes and agent nodes. The core nodes contain the state of the applications managed by ConductR, including where they are running, how many instances have been requested, and whether the number of requested instances has been met. The core node is also responsible for the decision-making related to the scaling up and

down of service instances. The agent nodes are responsible for the actual starting and stopping of the application processes.

Upon encountering a network partition, the segment of the network that contains the majority of the core nodes will continue running. The other instances of core nodes will automatically restart, waiting for the opportunity to rejoin the cluster. Once the network failure is remedied and the nodes are able to rejoin the cluster, the correct state of the applications managed by ConductR will be replicated to these core nodes.

When agent nodes encounter a network partition, each agent node will attempt to reconnect to a core node automatically. If the attempt to reconnect fails after a given period of time, the agent node will shut itself down, along with all the service processes that it was managing. Agent nodes stop the service processes to prevent a divergent application state that may occur during a network partition. Once all child processes have been shut down, the agent will attempt to automatically reconnect to all the core nodes it has previously known. Once it is able to rejoin with a core node, if the target number of application instances has not been met, the core node will instruct the agent to start new instances accordingly.

Let's test this behavior out. Restart the sandbox with `3` instances of ConductR core. Note the option `-n 3:3`, which indicates `3` core instances and `3` agent instances:

```
sandbox run 2.1.0 -n 3:3
```

Next, redeploy the Lagom Chirper example.

Maven:

```
./mvn-install
```

SBT:

```
sbt install
```

Then scale `Front-End` to `3` instances:

```
conduct run front-end --scale 3
```

After the scale request completes, the state should look similar to the following:

```
$ conduct info
ID               NAME                  VER  #REP  #STR  #RUN
acc2d2b          friend-impl            v1     3     0     1
bdfa43d-e5f3504  conductr-haproxy       v2     3     0     1
```

```
3349b6b          eslite                v1     3     0     1
188f510          chirp-impl            v1     3     0     1
f1c7210          load-test-impl        v1     3     0     1
93d0f25-44e4d55  front-end             v1     3     0     3
e643e4a          activity-stream-impl  v1     3     0     1
1acac1d          cassandra             v3     3     0     1
```

You can execute the following command to confirm the number of core node instances:

```
conduct members
```

The output you see should be similar to the following (i.e., there should be three core nodes running):

```
$ conduct members
UID          ADDRESS                STATUS  REACHABLE
-1775534087  conductr@192.168.10.1  Up            Yes
-56170110    conductr@192.168.10.2  Up            Yes
-322524621   conductr@192.168.10.3  Up            Yes
```

Next, execute the following command to confirm the number of core node instances:

```
conduct agents
```

The output should be similar to the following (i.e., there should be three agent instances running):

```
$ conduct agents
ADDRESS                                    OBSERVED BY
conductr-agent@192.168.10.1/client#165917  conductr@192.168.10.2
conductr-agent@192.168.10.2/client#-96672  conductr@192.168.10.2
conductr-agent@192.168.10.3/client#170693  conductr@192.168.10.3
```

Given that core and agent instances are bound to addresses that are address aliases for a loopback interface, the simplest way to simulate a network partition is to pause the core and agent instances. When the signal `SIGSTOP` is issued to both core and agent instances, they will be paused and effectively frozen in execution. From the perspective of the other core and agent nodes, the frozen core and agent nodes have become unreachable, effectively simulating a network partition from their point of view.

In order to demonstrate ConductR's self-healing capability for network partitions, let's simulate a network partition. To do this, first pause the agent instance listening on `192.168.0.3` by executing the following shell command:

```
pgrep -f "conductr.ip=192.168.10.3" | xargs kill -s SIGSTOP
```

Similarly, let's pause the core instance listening on `192.168.0.3`:

```
pgrep -f "conductr.agent.ip=192.168.10.3" | \
xargs kill -s SIGSTOP
```

Monitor the member state by issuing a `watch` on `conduct members`. For those running on macOS, the `watch` command can be installed via `brew` using `brew install watch`. Alternatively, simply issue the `conduct members` command repeatedly:

```
watch conduct members
```

After a full minute or two, you'll see that the member on `192.168.10.3` has become unreachable (i.e., `REACHABLE` is `No`):

```
UID          ADDRESS                STATUS  REACHABLE
-1775534087  conductr@192.168.10.1  Up            Yes
-56170110    conductr@192.168.10.2  Up            Yes
-322524621   conductr@192.168.10.3  Up             No
```

Eventually, the member on `192.168.10.3` will be considered down by other members and will be removed from the member list:

```
UID          ADDRESS                STATUS  REACHABLE
-1775534087  conductr@192.168.10.1  Up            Yes
-56170110    conductr@192.168.10.2  Up            Yes
```

Similarly, start a `watch` on `conduct agents`, or execute `conduct agents` repeatedly. This enables you to observe the agent being removed from the cluster:

```
watch conduct agents
```

After at least one minute, you will see that the agent on `192.168.10.3` can no longer be observed by any remaining member:

```
ADDRESS                                    OBSERVED BY
conductr-agent@192.168.10.1/client#165917  conductr@192.168.10.2
conductr-agent@192.168.10.2/client#-96672  conductr@192.168.10.2
conductr-agent@192.168.10.3/client#170693
```

Eventually, the agent on `192.168.10.3` will be considered down by other members and will be removed from the member list:

```
ADDRESS                                    OBSERVED BY
conductr-agent@192.168.10.1/client#165917  conductr@192.168.10.2
conductr-agent@192.168.10.2/client#-96672  conductr@192.168.10.2
```

Once this occurs, issue `conduct info` to see the state of our cluster. The `#REP` column indicates the replicated copy of the bundle file has been reduced from `3` to `2` due to the missing core node indicated by the conduct members. The `#RUN` column of the `front-end` has been

reduced from 3 to 2 due to the missing agent indicated by the conduct agents:

```
$ conduct info
ID              NAME                 VER  #REP  #STR  #RUN
acc2d2b         friend-impl           v1     2     0     1
bdfa43d-e5f3504 conductr-haproxy      v2     2     0     1
3349b6b         eslite                v1     2     0     1
188f510         chirp-impl            v1     2     0     1
f1c7210         load-test-impl        v1     2     0     1
93d0f25-44e4d55 front-end             v1     2     0     2
e643e4a         activity-stream-impl  v1     2     0     1
1acac1d         cassandra             v3     2     0     1
```

Now let's unfreeze both the core and agent instance on `192.168.10.3`:

```
pgrep -f "conductr.agent.ip=192.168.10.3" | xargs kill -s SIGCONT
pgrep -f "conductr.ip=192.168.10.3" | xargs kill -s SIGCONT
```

When you do this, the core and agent instance on `192.168.10.3` will realize that they have been split from the cluster, and will automatically restart.

Eventually, the `conduct members` command will indicate that a new core instance on `192.168.10.3` has rejoined the cluster. Below, the new core instance is indicated by the new `UID` value of `-761520616`, while the previous core instance had a value of `-322524621`. Note that you will observe different UID values on your screen than what I have show here:

```
UID          ADDRESS                 STATUS  REACHABLE
-1775534087  conductr@192.168.10.1  Up            Yes
-56170110    conductr@192.168.10.2  Up            Yes
-761520616   conductr@192.168.10.3  Up            Yes
```

Similarly, the `conduct agents` command will eventually indicate that the restarted agent has rejoined the cluster:

```
$ conduct agents
ADDRESS                                    OBSERVED BY
conductr-agent@192.168.10.1/client#165917  conductr@192.168.10.2
conductr-agent@192.168.10.2/client#-96672  conductr@192.168.10.2
conductr-agent@192.168.10.3/client#170693  conductr@192.168.10.2
```

And finally, the state of the cluster is restored as before with the `#REP` value of `3`, and the `front-end` has recovered back to `3` instances:

```
$ conduct info
ID              NAME                 VER  #REP  #STR  #RUN
acc2d2b         friend-impl           v1     3     0     1
```

```
bdfa43d-e5f3504  conductr-haproxy       v2     3     0     1
3349b6b          eslite                v1     3     0     1
188f510          chirp-impl            v1     3     0     1
f1c7210          load-test-impl        v1     3     0     1
93d0f25-44e4d55  front-end             v1     3     0     3
e643e4a          activity-stream-impl  v1     3     0     1
1acac1d          cassandra             v3     3     0     1
```

You have just observed ConductR's self-healing capability for network partitions. Automatic, self-healing recovery from network partitions absolves operations from not only having to detect when a network partition has occurred, but also from determining which nodes to keep and which must be restarted in response to the failure.

In this chapter you deployed the Lagom Chirper sample application using Lightbend Enterprise Suite and the generated installation script. You induced failures upon the cluster, killing processes and partitioning networks to observe ConductR's resilience and self-healing capabilities. See the project *deploy.md* for more information and other recovery examples to run.

CHAPTER 4
Conclusion

> To infinity, and beyond!
>
> —Buzz Lightyear, *Toy Story*

In this report we have examined the application of the Reactive principles to tools and services for automating deployment, scaling, and management of containerized microservice applications across a distributed cluster.

We've identified important traits that you should look for across the application delivery pipeline. The deployment platform must be designed for distributed operation, and thus, like our own applications, the platform should also be Reactive.

You should seek to avoid introducing strongly consistent data management services into the critical layers of your service orchestration, service lookup, and overall cluster management. Teams need to be able to scale containers horizontally across numerous nodes with agility and confidence. Simply stated, it should be enjoyable to deliver your services to production.

From developer test to production delivery, there should be a simplification of the interfaces and concepts presented to users, developers, and operators alike. We should evaluate the developer library support so that our teams can focus on business problems and not on implementation details such as how to manage peer-node clusters. We need a continuous delivery pipeline to automate the process of deploying in order to enable teams to test and deliver updates quickly and reliably. The selection of standards, such as the OCI

container image and runtime, continue to aid in mitigating vendor lock-in. Reactive microservices are best delivered using Reactive deployment tools that are both operations and developer friendly.

Also in this report you deployed a Reactive application to a Reactive delivery and deployment platform. You deployed the sample application Chirper using Lightbend Enterprise Suite. I encourage you to continue experimenting with and exploring the deployment. In this report you induced some failures so that you could observe the resilience and self-healing features, firsthand. Many additional failure scenarios exist, and you're welcome to test other use cases. Be certain to see the the project repository on GitHub for other test cases.

Not that many years ago, the smallest of development test clusters required a four-posted server rack to hold all the parts. Today, presenters now only need to bring a small box with several Raspberry Pi boards and some switches to live-demonstrate a clustered solution. In Chapter 3 we ran multiple instances of an application service using Lightbend Enterprise Suite in a production-like cluster environment. We tested our clusters by inducing failures, including a network partition, so that you can be assured of its resilience and observe its self-healing.

Efforts continue to further simplify the task of delivering scalable and resilient services with agility. Delta State Replicated Data Types, for example, reduce the amount of state that needs handling when performing updates across the clustered CRDT. We are likely to see new ways of testing emerge as it becomes easier to define and restore not only the collection of container services that compose an application, revision information, and so on, but also the state of the persistent actors in the running services. It is conceivable that, like algorithmic traders testing new trading strategies against replays of market data, we might apply Lineage-Driven Fault Injection and machine learning to the events, commands, and facts from our own systems to train them to be more resilient. Intelligent auto-scaling utilizing self-tuning and predictive analysis will not be far behind.

Throughout this three-part series of reports on Reactive microservices, we've seen how the Reactive principles are represented in the designing, development, and deployment of microservices. As data and data-driven software becomes essential to the success of organizations, they are adopting the best practices of software development. It is the innovation, hardening, and re-architecting of over 40

years of research and real-world usage that bring us Reactive microservices. We know that failures will happen, so you must embrace them by considering them up front. When you do, you view deployment in a whole new light. Instead of a weight that must be carried, deployment can become the exciting delivery of the new, faster, and better versions of your software to your customers and subscribers. Flexible and composable, your deployment platform becomes a highly effective weapon in the rapid telemetry, high-demand markets that organizations compete in today. We hope that this series has helped you better understand the critical importance of using a Reactive deployment system when delivering Reactive applications.

This concludes this Reactive microservice series. It has been our sincere pleasure to introduce you to Reactive microservices and the joys of a fully Reactive deployment. We hope that the Reactive strategies and tools presented here are just the beginning of your Reactive journey. Happy travels!

About the Author

Edward Callahan is a senior engineer at Lightbend. Ed started delivering Java and JVM services into production environments back when NoSQL databases were called object databases. At Lightbend he developed and deployed early versions of Reactive Microservices using Scala, Akka, and Play with prerelease versions of Docker, CI jobs, and shell scripts. Those "Sherpa" services went on to become the first production deployment using the Lightbend Enterprise Suite. He enjoys being able to share the joys of teaching and learning while working to simplify building and delivering streaming applications in distributed computing environments.

Acknowledgments

Ed would like to especially thank Christopher Hunt, Markus Jura, Felix Satyaputra, and Jason Longshore for their contributions to this report. This publication was a team effort and would not have been possible without their contributions.