White paper
Cloud Transformation
and IT Modernization

CGI
Experience the commitment®

Breaking the monolith application:
A deep dive into replatforming

Abstract

Our previous whitepaper, "Rebuilding and replatforming legacy applications" dove into CGI's two-step approach for modernization: (1) replatforming existing applications from bare-metal or virtual machines to containerized target environments and (2) gradually eliminating complex applications by rebuilding their functionality as native cloud services. The requirements are clear: applying a streamlined approach is essential to effectively building and leveraging cloud native services.

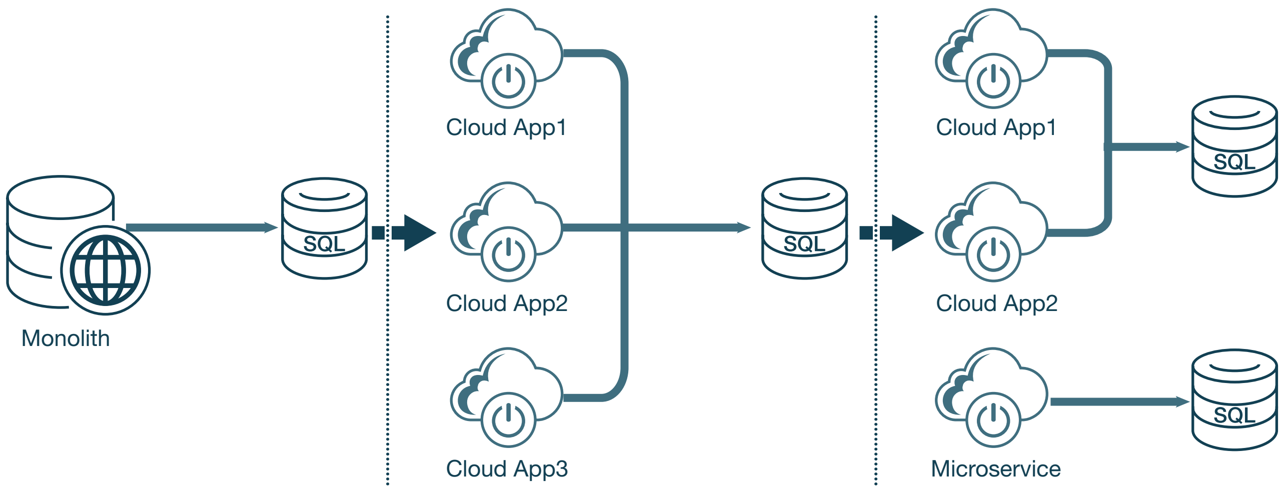
Curious how we break it down further? The CGI cloud-native team has helped many clients transform their monolith applications into smaller cloud native applications that can scale and benefit from a cloud platform in a relatively small timeframe. This whitepaper dives deeper into CGI's approach for replatforming monolith applications with a look into our cloud transformation approach. We share the details of our methodology, highlighting the strategies, tooling, and platforms that have helped our projects succeed.

This whitepaper describes a very detailed process of how cloud transformation was implemented in a specific project. Therefore, in addition to the used procedure, the deployed applications and partner companies are also mentioned.

Defining the terms: Cloud-native applications versus microservices

A microservice by default is often built as a cloud-native application. However, not all cloud native applications are necessarily microservices. A mature cloud-native application should adhere to the generally-accepted Twelve-Factor Application principles and it is not coupled to a specific container. In this type of mature application, all backing services are detached and bound at runtime; and it is light weight with a small startup and shutdown time. Because of these characteristics, a cloud-native application can benefit from a cloud container platform by being scalable and deployable independently from other applications.

A microservice has some inherent complexity. Not every application is a good fit to be turned into a microservice. By definition, a microservice should not share anything with other applications, such as a database or common libraries. Transitioning from a monolith application directly into a microservice model is a high risk and high effort undertaking. In many cases, the ideal transformation path for a monolith application is to break it into smaller cloud-native applications, and later on, if applicable, into microservices. This diagram depicts this path:



Our transformation recipe

Breaking a monolith application is not a trivial exercise. Most team members may perceive it as daunting. After going through multiple application transformation projects at CGI, we have created a repeatable transformation approach to mitigate risk by addressing the monolith application in iterative stages. With the exception of Stage 1, all stages can and should be executed in parallel. The stages are as follows:

- Stage 1 – Make the monolith work outside the current container
- Stage 2 – Make the monolith work in the cloud
- Stage 3 – Break the monolith into smaller applications or micro services
- Stage 4 – Virtualize ("lift and shift") internal hosted services

Our transformation journey

Let's walk through a hypothetical transformation journey with an application based on CGI's collective recent projects. Here's what we are working with: The initial monolith application consists of a public-facing, Java-based web application. This monolith hosts multiple REST and SOAP based APIs. The primary application database is a Microsoft SQL Server database. It uses Apache ActiveMQ for message queuing.

Our goal for this hypothetical transformation is to break the monolith into smaller cloud-native applications and deploy them using Pivotal Platform hosted on Google Cloud Platform (GCP).

Stage 1 – Make the monolith work outside the current container

Codebase (Factor I[1])

The first step is to get the codebase migrated into the desired source control repository. A new branch needs to be forked for the cloud transformation effort. In most cases, ongoing development cannot be halted and thus a code synchronization strategy will be needed. It helps if the same source control system is used for both original and transformed code (e.g. GIT to GIT). However, that is not always an option. For instance, the current source system may be SVN (Subversion) and the desired target source system may be GIT. In that case there are bridges such as git-svn that can help merge the code.

Dependencies (Factor II)

Dependencies from your code should be available through an internal artifact repository manager, such as Nexus or Artifactory. As part of this effort you may need to make all dependencies available through your corporate artifact repository manager.

You also need to externalize dependencies from your code, if not already done. In Java applications, this can be accomplished using Gradle build files or Maven POM (Project Object Model) files. Gradle allows you to easily convert Maven POM files to Gradle build files. Both Gradle and Maven have equivalent capabilities. We tend to prefer Gradle, as it uses a more compact format (Groovy based) than Maven (XML based).

Decouple from current container

A big step in this stage is to make the monolith application work outside the current container. In our example application, the application is hosted using Apache Tomcat which makes it relatively easy to decouple from the container. We introduced Spring Boot and embedded Apache Tomcat as a dependency. Additionally, we had to convert configuration from the application's web.xml file (e.g. URL mappings) into equivalent Spring annotations.

In other projects, the monolith may be hosted in WebSphere or WebLogic Application Servers. A full JavaEE Application Server provides more container services, sometimes proprietary, which could make it harder to detach from that container. In the case of IBM, they provide tools that analyze existing applications and provide a list of changes needed. The WebSphere Liberty profile is a good transition path to the cloud, as it runs in most cloud containers, including Pivotal Platform and Red Hat OpenShift.

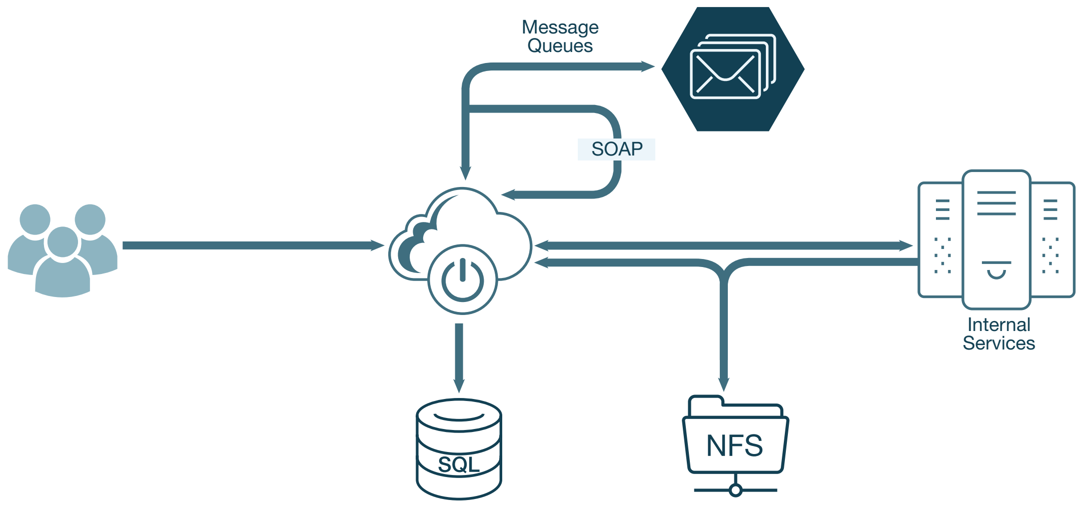
[1] From the Twelve-Factor Application Principles

Externalize configuration (Factor III)

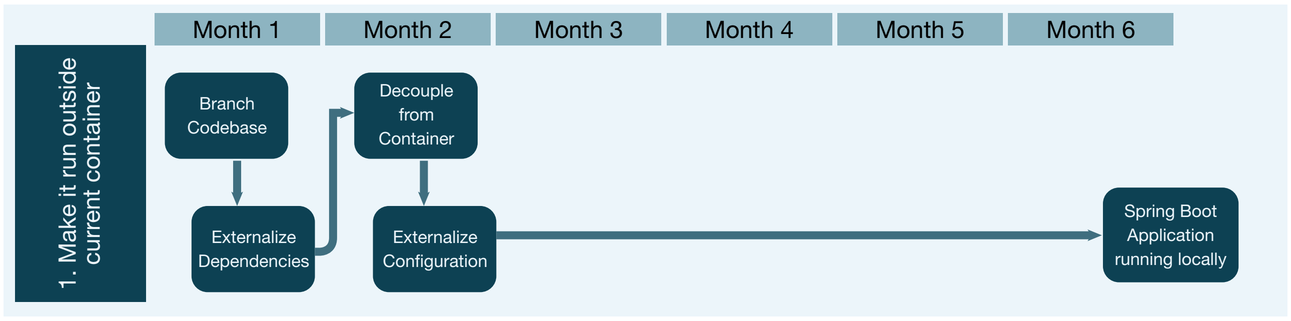
In the 'Externalize Configuration' stage, we externalize property values that are stored in the web.xml file into application YML (Yet another Markup Language) files. There are often code changes needed to make the legacy classes worked with the Spring injected configuration (read from YML files).

Stage 1 complete

At the end of stage 1, you should have a Spring Boot monolith application running locally. The application could be deployed to a cloud container and partially functional, as long as most of the backing services (e.g. database) are reachable from the cloud platform.



This flow chart depicts a typical timeframe for this stage (1-2 months):


Month 1
Month 2
Month 3
Month 4
Month 5
Month 6
1. Make it run outside current container
Branch Codebase
Externalize Dependencies
Decouple from Container
Externalize Configuration
Spring Boot Application running locally

Stage 2 – Make the monolith work in the cloud

Replace container provided services

An Application Server provides value added services. By decoupling from the application server, we now need to replace those services provided. An example of these services is database connection pooling. To replace this service, we introduced HikariCP, a popular library used for database connection pooling in detached containers.

Externalize configuration – Part 2

Configuration was first externalized into application YML files in stage 1. We now need to make them available to our application in the cloud. We can accomplish this by using Spring Cloud Config and we configure the Spring Cloud Config Server to read environment specific files from a GIT repository. At runtime, the application instances connect to the cloud and retrieve the configuration file as part of the Spring boot initialization.

Backing services (Factor IV)

In a cloud-native application, all backing services (e.g. database, message queuing system) need to be provided to the application and bound at runtime. Changing the application to treat services this way is generally not a large effort; however, it can be difficult to make the services available to the cloud platform, especially if they are not available through the corresponding cloud platform service catalog. These are some examples of backing services and strategies for making them available as attached resources in the cloud:

- **Web session storage**
  In most legacy applications, web session is stored in memory and user sessions are preserved by routing the user to the same server (using "sticky" sessions). In a cloud-native application, the web session must be offloaded from memory. A popular solution for offloading web session is Redis. Redis provides a highly available and high-performance session store which can be easily integrated with Spring. Furthermore, Redis is available in the Pivotal Platform's service marketplace and can be bound to applications at runtime. Another alternative to Redis for web session storage is Pivotal Cloud Cache (PCC).

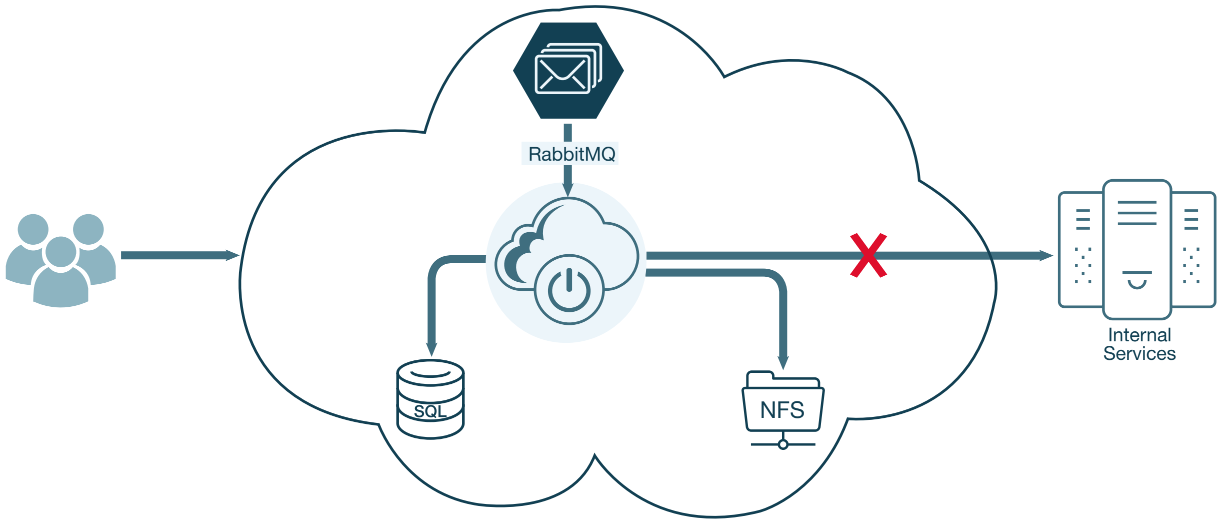
- **Database**
  There are multiple databases that can be provisioned via the Cloud Platform service catalog. For instance: MySql, MongoDB, PostgreSQL are available in PCF's Service Marketplace. Due to the licensing model, Microsoft SQL Server is not currently available in PCF's marketplace, hence we had to make the database available in a GCP virtual instance.
- **File share**
  File share access is not the most ideal way to transfer info in the cloud, but most legacy applications rely on it and replacing it can be a large effort. Fortunately, there are a few options to make file share access work in the cloud. One is simply to make NFS (Network File System) available to the cloud. This can be an NFS hosted in multiple virtual instances in the cloud.[2] As a general rule for file sharing in cloud native applications we recommend using an object store instead of leveraging NFS file shares via the Volume service on the Pivotal Platform.

Build and deploy pipelines (Factor V - Build, release, run)

At this point, we have an application ready to run in the cloud in multiple instances. Next, we need a way to build our code and deploy it to the cloud. Jenkins is a popular CI/CD tool (Continuous Integration/Continuous Deployment) tool for accomplishing this. Along with Jenkins, we use Concourse for pipelines as a service. We leverage Jenkins pipelines to trigger the build (upon commits to develop branch in our git repository) and also to deploy to development, QA, and eventually production. Code should be built once and the same binary should be deployed to different environments.

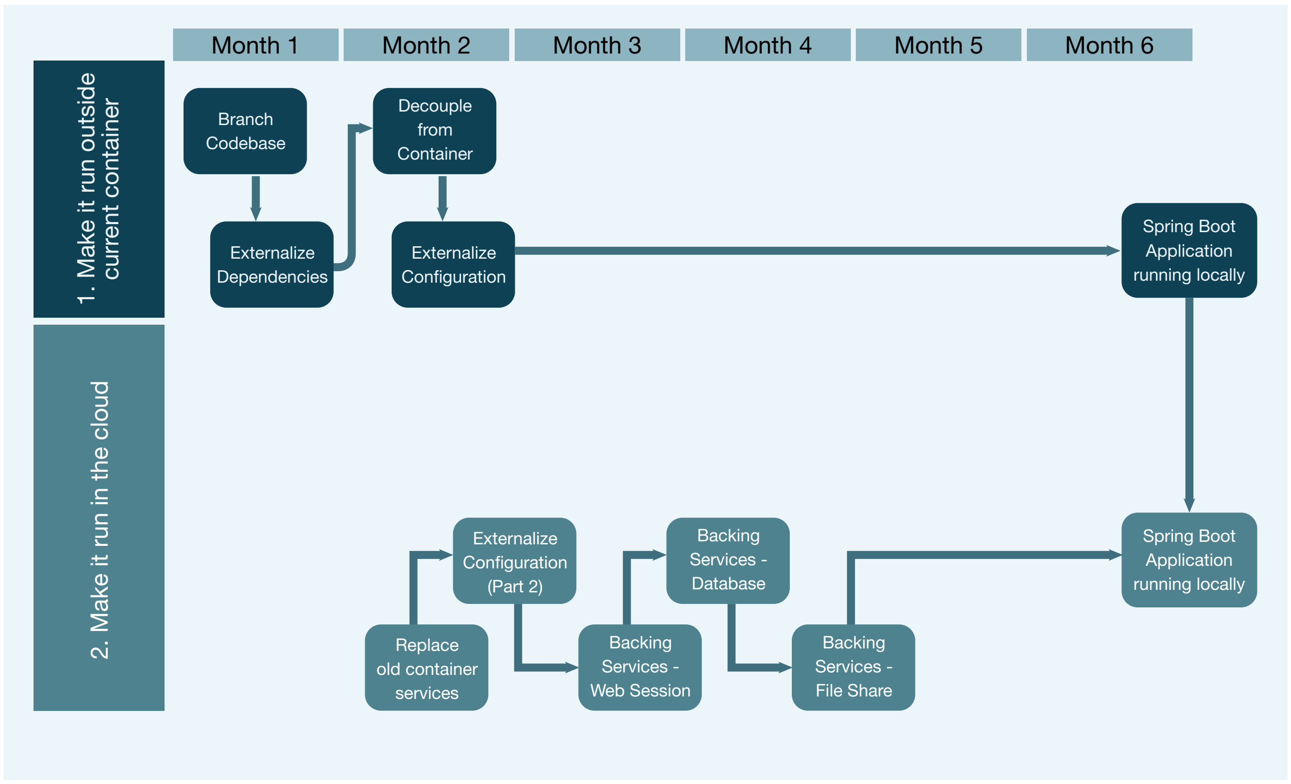
Stage 2 complete

At the end of this stage, your application should be running in the cloud using multiple instances and be almost fully functional, depending on whether all internal services used by the application are available in the cloud.



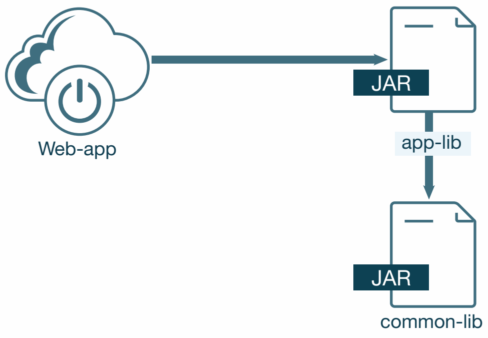
[2] We explored the use of the NFS service tile available for the Pivotal Platform, however at the time of writing, this service was still in the experimental phase and could not be used for go-live. Instead, we relied on the use of our cloud platform specific capability, i.e. Google Cloud Storage Buckets. Changing the code to use buckets was not a significant effort, but that may vary depending on your application.

Our updated swim lane diagram shows how tasks can be executed in parallel:


Month 1
Month 2
Month 3
Month 4
Month 5
Month 6
1. Make it run outside current container
2. Make it run in the cloud
Branch Codebase
Externalize Dependencies
Decouple from Container
Externalize Configuration
Spring Boot Application running locally
Replace old container services
Externalize Configuration (Part 2)
Backing Services - Web Session
Backing Services - Database
Backing Services - File Share
Spring Boot Application running locally

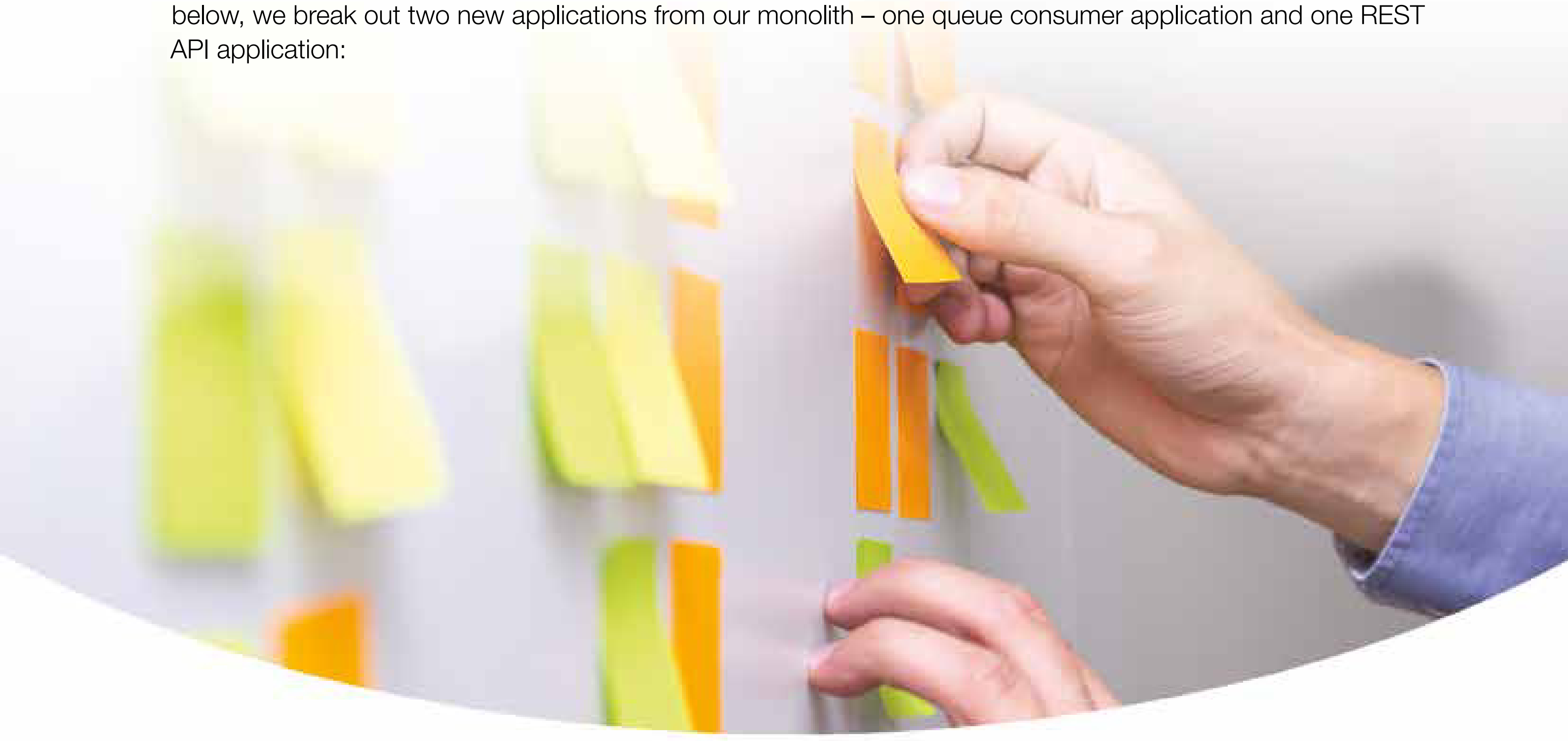
Stage 3 – Break monolith into smaller applications/microservices

A monolith application normally consists of one application that contains multiple custom application libraries:

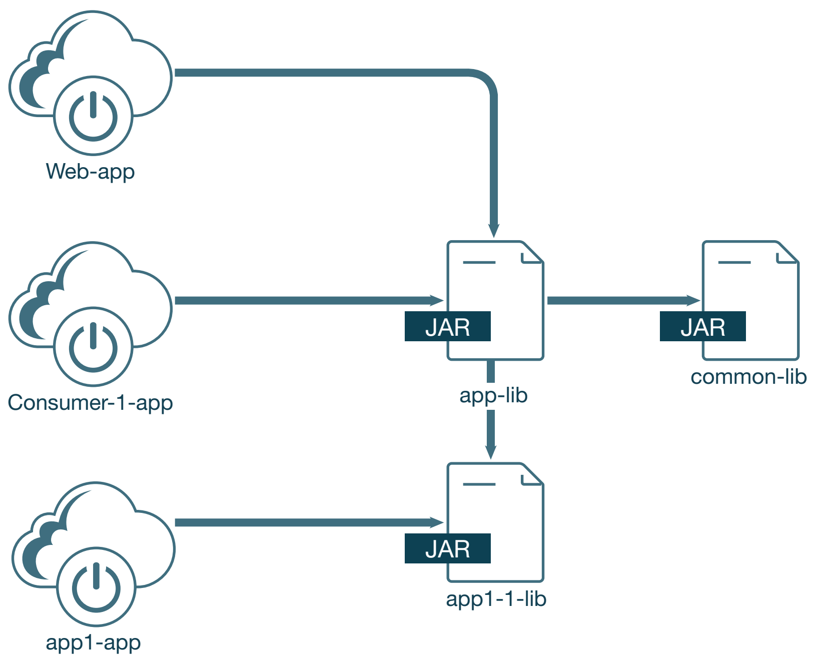


Even if your end goal is not to rely on microservices, we recommend following the Application Continuum approach[3] for breaking your monolith applications into applications and components.

The first step is identifying candidate applications to become new applications. Inspect all of the entry points to your application: web UI, REST APIs, message queue consumers and scheduled tasks. Those entry points are good candidates to become stand-alone applications. In addition to the Application Continuum, there are multiple strategies for microservices. We also extensively employ Domain Driven Design (DDD) techniques such as Event Storming and Domain Storytelling. During this process, also identify ways to break shared libraries into smaller libraries to identify new code boundaries and possible candidates for microservices. In the diagram below, we break out two new applications from our monolith – one queue consumer application and one REST API application:

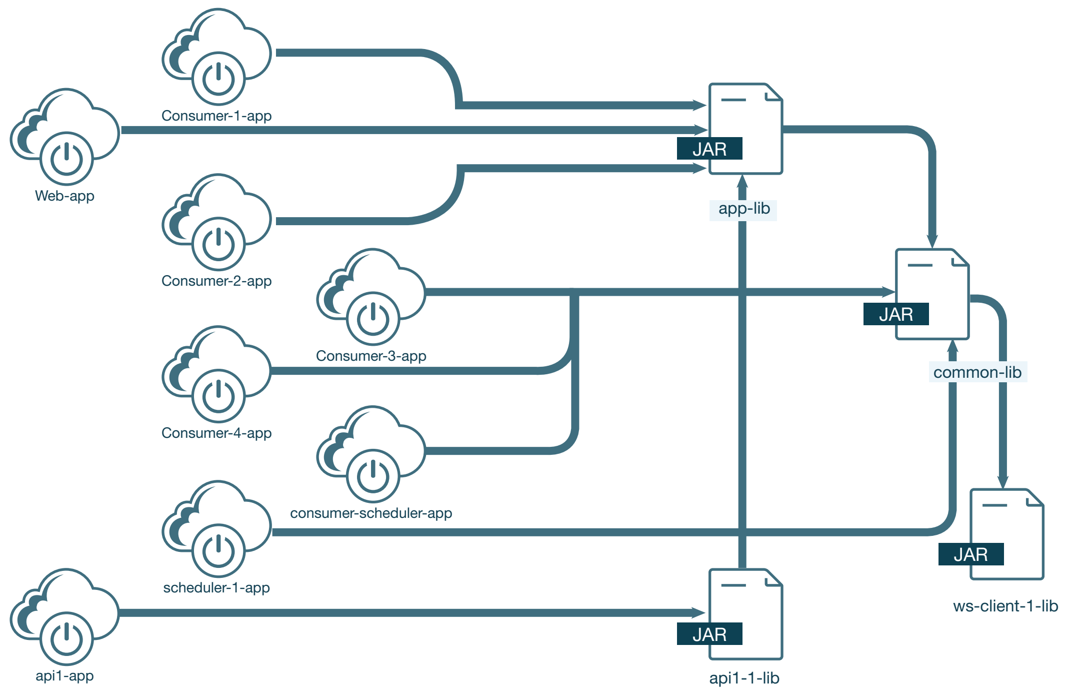


[3] https://www.appcontinuun.io



These 2 new applications are also implemented as Spring Boot applications and can be deployed independently from the main web application. They can also be scaled independently. The new applications have small startup and shutdown times because they are initializing less components. They are also faster to deploy as the application binaries size is smaller.

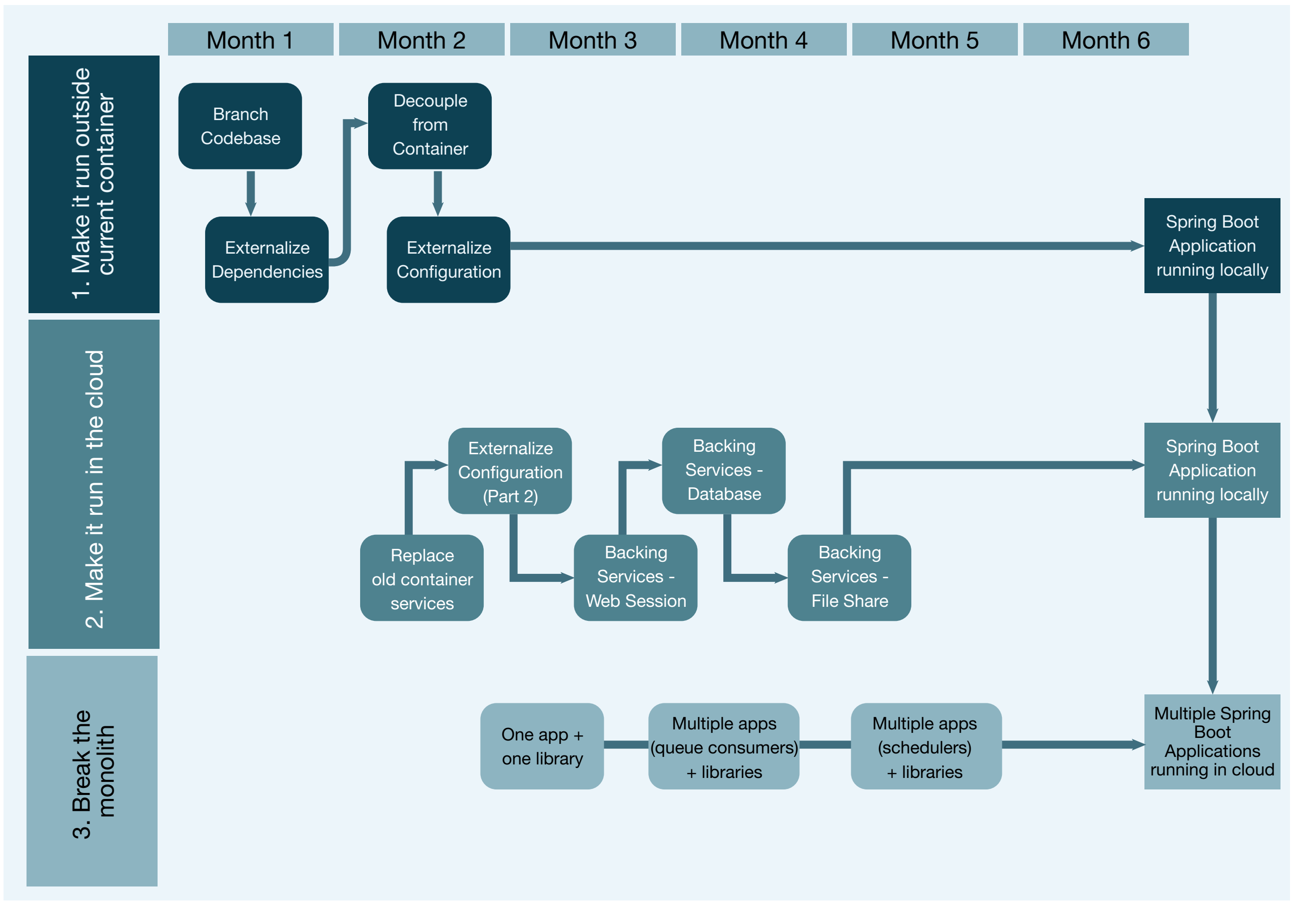
Continuing through this process, we end up with a total of 8 applications: 4 queue consumer apps, 1 REST API application and 2 scheduled tasks applications. Our monolith application is now much smaller, and it is exclusively a web application with no unnecessary entry points.

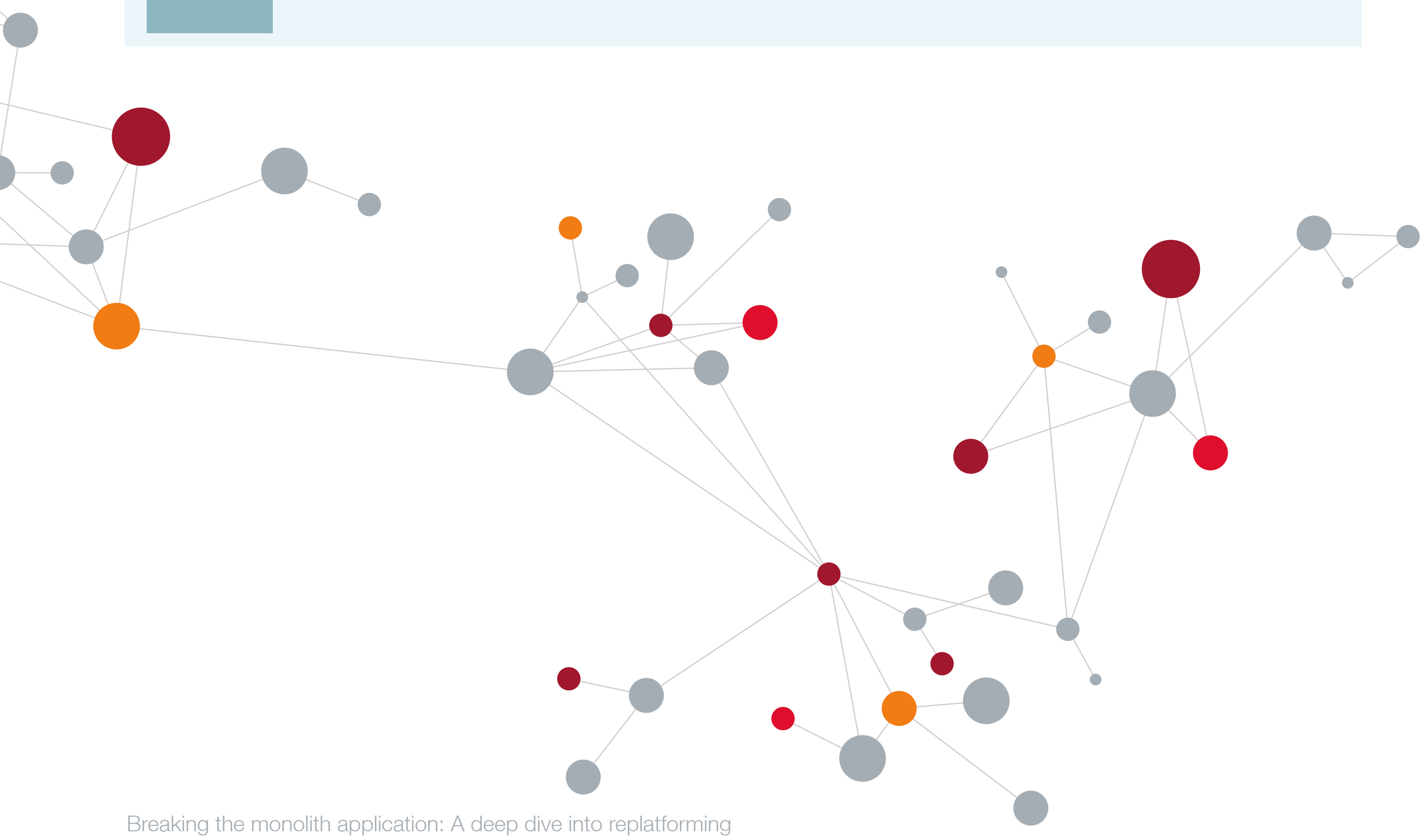


These new, smaller applications are all cloud-friendly and can be released and scaled independently from each other. However, they still share common libraries and a common database, hence they cannot be considered microservices. If a defect is found in one of the common libraries, it may force you to redeploy all applications using that library.

This transformation can be accomplished in a relatively short timeframe, allowing you to move to the cloud quicker and enabling your organization to release sooner and more frequently. The libraries can be gradually broken into smaller libraries and eventually become microservices.

Our transformation swim lane shows that at the end of stage 3, we now have multiple Spring Boot applications running in the cloud that could be released assuming that all external systems are reachable in the cloud. It also shows the parallelism within the 3 stages.


Month 1
Month 2
Month 3
Month 4
Month 5
Month 6
1. Make it run outside current container
2. Make it run in the cloud
3. Break the monolith
Branch Codebase
Externalize Dependencies
Decouple from Container
Externalize Configuration
Spring Boot Application running locally
Replace old container services
Externalize Configuration (Part 2)
Backing Services - Web Session
Backing Services - Database
Backing Services - File Share
Spring Boot Application running locally
One app + one library
Multiple apps (queue consumers) + libraries
Multiple apps (schedulers) + libraries
Multiple Spring Boot Applications running in cloud



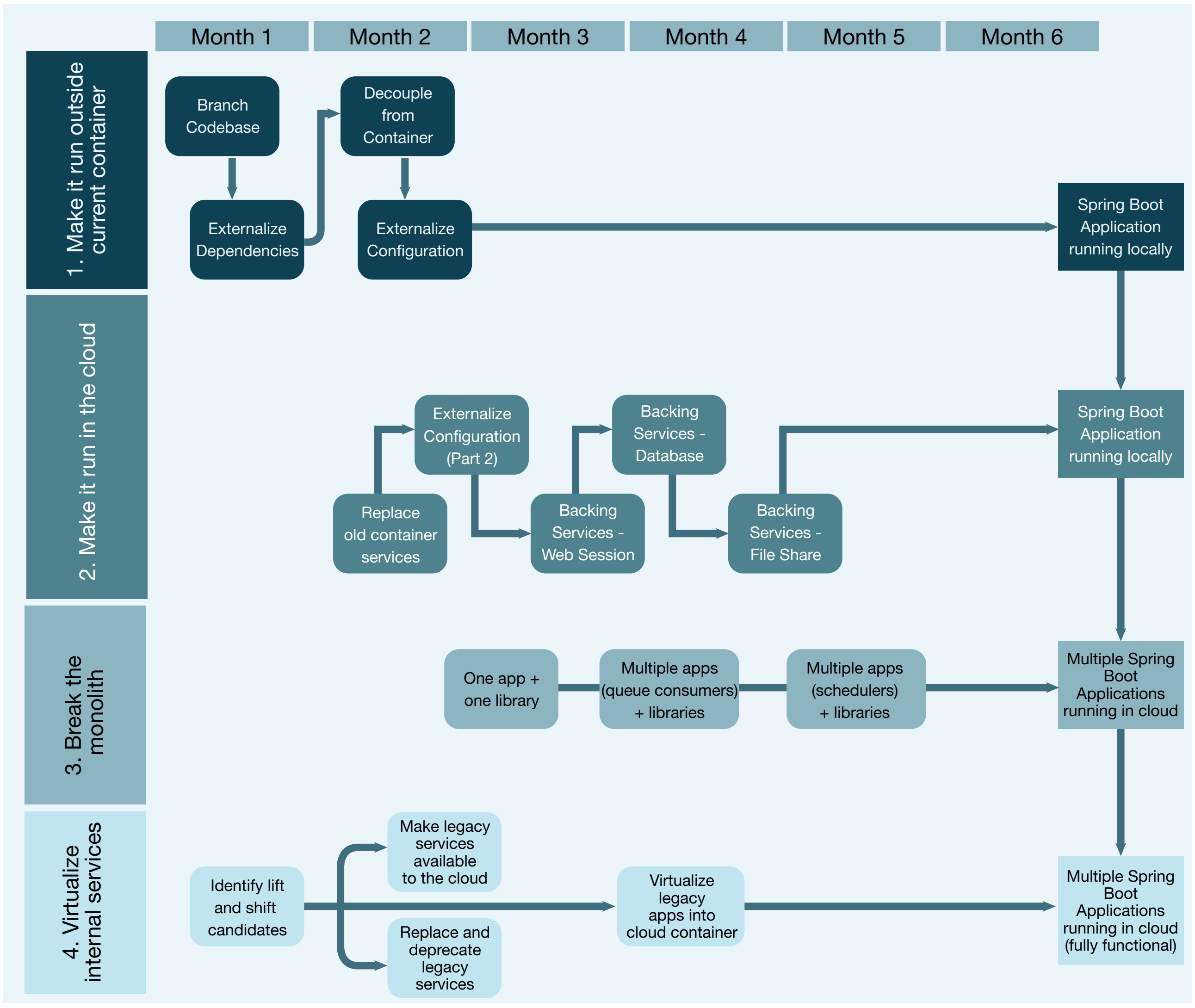
Stage 4 – Virtualize/lift and shift internal hosted services

This stage is often the least exciting one, albeit an important one. Your monolith may depend on enterprise systems that are not going to be moved to the cloud. It may also leverage packaged applications. It is critical to identify and plan a strategy for these systems as early as possible in the project.

Because this stage typically requires the most elapsed time, it is important to start this stage early in parallel with other stages, as it may require infrastructure changes or impact systems that you have less control over. The first step in this stage is to identify these shared applications and determine one of the following strategies:

1. Can they be virtualized (“lift and shift”) in the cloud? This will be the case when all the dependent applications are hosted in the cloud.
2. Can the shared application become reachable from the cloud? This can be a good option for applications that are shared between cloud and non-cloud hosted applications.

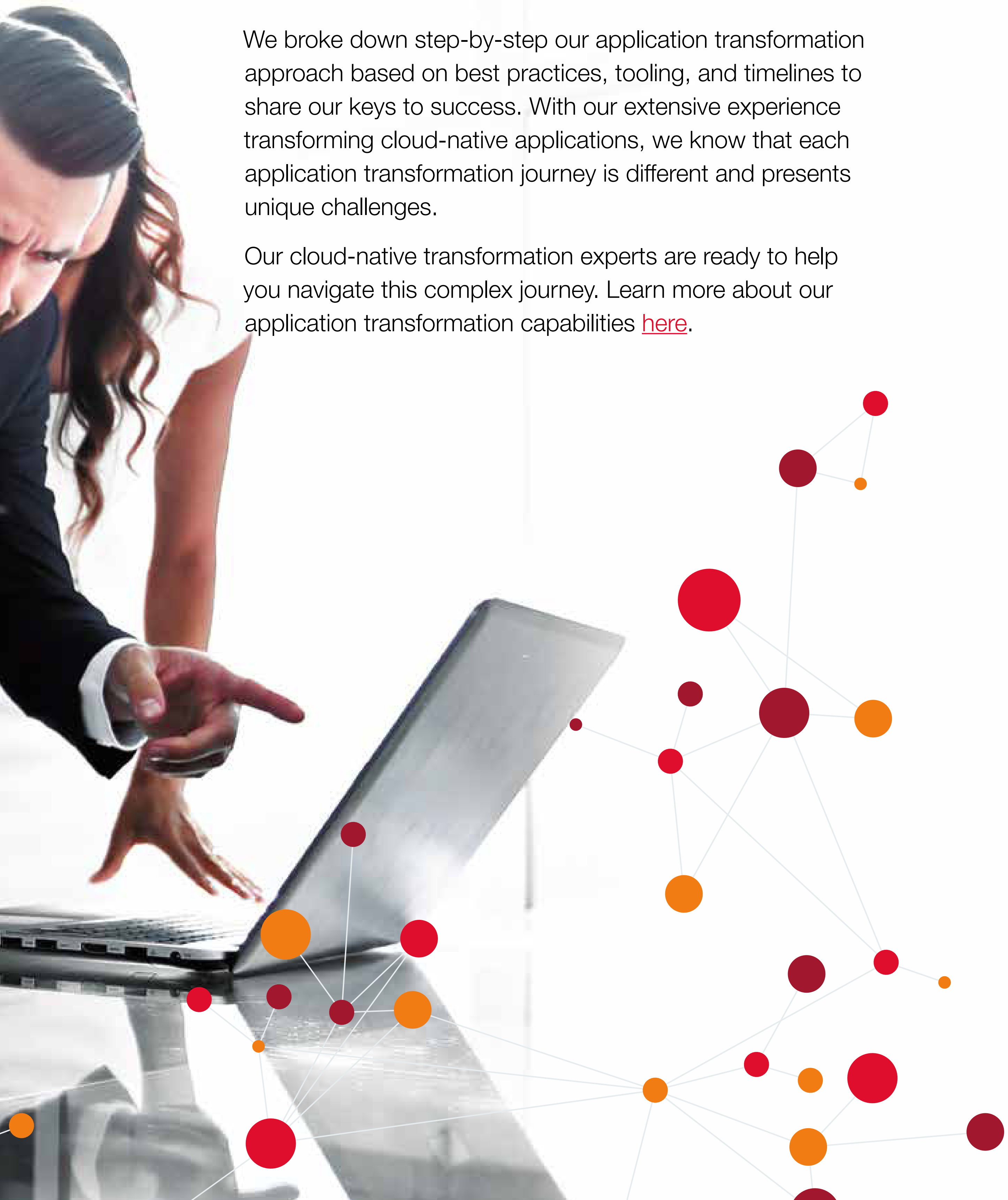
Once all dependent applications are made available from the cloud platform, our new transformed cloud applications will be fully functional in the cloud and will be ready to be released.


Month 1
Month 2
Month 3
Month 4
Month 5
Month 6
1. Make it run outside current container
Branch Codebase
Externalize Dependencies
Decouple from Container
Externalize Configuration
Spring Boot Application running locally
2. Make it run in the cloud
Replace old container services
Externalize Configuration (Part 2)
Backing Services - Web Session
Backing Services - Database
Backing Services - File Share
Spring Boot Application running locally
3. Break the monolith
One app + one library
Multiple apps (queue consumers) + libraries
Multiple apps (schedulers) + libraries
Multiple Spring Boot Applications running in cloud
4. Virtualize internal services
Identify lift and shift candidates
Make legacy services available to the cloud
Replace and deprecate legacy services
Virtualize legacy apps into cloud container
Multiple Spring Boot Applications running in cloud (fully functional)

Are you ready to move your monolith to the cloud?

We broke down step-by-step our application transformation approach based on best practices, tooling, and timelines to share our keys to success. With our extensive experience transforming cloud-native applications, we know that each application transformation journey is different and presents unique challenges.

Our cloud-native transformation experts are ready to help you navigate this complex journey. Learn more about our application transformation capabilities here.



CGI

About CGI

Founded in 1976, CGI is among the largest IT and business consulting services firms in the world. Operating across the globe, CGI delivers end-to-end capabilities, from strategic IT and business consulting to systems integration, managed IT and business process services and intellectual property solutions, helping clients achieve their goals, including becoming customer-centric digital enterprises.

CGI was pleased to receive the 2019 EMEA Systems Integrator of the Year for Customer Impact by Pivotal Software, Inc. The Customer Impact award recognizes CGI for its leadership in driving customer success and scale with the Pivotal Platform. In 2018, CGI received the 2018 AMER Systems Integrator of the Year for Platform Scale award from Pivotal. The Platform Scale award recognizes CGI for its technical expertise and leadership in helping to promote the success of the Pivotal Platform as part of our application transformation engagements.

CGI Deutschland B.V. & Co. KG
Leinfelder Straße 60
70771 Leinfelden-Echterdingen
T.: +49 711 72846-0
E-Mail: info.de@cgi.com
www.de.cgi.com

brandeins
/thema
b
2020
Heft 13
Beste
IT-Dienstleister