White Paper: **April 2023**

Automate Middleware Migration to Open Source Frameworks with Ease

vmware®

Table of contents

With years of helping enterprises move to open source frameworks such as Spring, VMware Tanzu Labs has streamlined the migration process with tools and recipes, or repeatable patterns that can be used for multiple applications and workloads.

Introduction

In this highly competitive and innovative era, many organizations are desperate to move faster and find ways to be even more efficient. Across industries, enterprises are realizing the importance of migrating off legacy middleware to modern, open source frameworks to access the flexibility and agility they provide, such as significantly reducing costs and management needs.

At one point, middleware such as enterprise service buses (ESBs) played a critical role in enabling applications to connect and communicate with one another. Although that sounds great in theory, many organizations tend to run into problems with ESBs over time. With the low-code nature of service buses, it's easy for teams to wire up their logic using drag-and-drop GUIs, causing the logic to become extremely difficult to test and maintain. In addition, when the enterprise service bus needs to be upgraded to a new version, it might require a new syntax for the applications running on it. With each of these challenges, the ESB has evolved into the new monolith, causing many organizations to want open source frameworks.

With years of helping enterprises move to open source frameworks such as Spring®, VMware Tanzu Labs™ has streamlined the migration process with tools and recipes, or repeatable patterns that can be used for multiple applications and workloads.

In this white paper, we share the best practices that we use to help our customers successfully automate their middleware migration initiatives, and share an example based on a recent engagement with a large financial services organization.

Enterprise service buses vs. open source frameworks

Let's discuss the challenges of enterprise service buses and how open source frameworks can solve them.

Challenges of enterprise service buses

ESBs are considered the new monolith because of some of the challenges they pose, including:

- High recurring licensing costs and vendor lock-in
- Difficulty in testing and maintaining logic created with drag-and-drop ESB designers
- High cognitive load of proprietary frameworks interfering with developer productivity
- New version upgrades often require syntax updates for older services

Benefits of open source frameworks

Migrating to open source frameworks, such as Spring, addresses the challenges of enterprise service buses. Some of the benefits of open source frameworks include:

- Reduce licensing costs
- Relieve cognitive load on developers
- Reduce need for specialty skillsets
- Lower the amount of development frameworks
- Gain the ability to run services anywhere
- Build modern applications and services
- Attract and maintain talented developers

The benefits of migrating to open source frameworks are undeniable, but the process of migration itself can present its own obstacles.

Considerations when building a migration strategy

Until recently, it was generally accepted that if an organization needed to change source code to move between technology stacks, it would require significant manual effort. Migrating legacy middleware workloads to open source frameworks manually can be tedious and time-consuming—having to catalog existing services, define the transformation pattern for each type of service, and eventually code each of these changes.

VMware Tanzu Labs helps our customers automate their migration process and realize outcomes even faster using modern techniques and sophisticated tooling.

However, before making the immediate decision to automate the migration process, we believe there are a few key considerations to keep in mind:

- How can you be sure that developers will use the automated tools that are created?
- How can you be sure that developers will accept and be able to understand the code generated?
- How can you be sure that the output quality of the tools meets your company's quality of standards?

In addition to these considerations, we believe that large-scale migration exercises such as these should be approached as a community effort between relevant teams. By establishing a community of practice and building a shared set of resources, organizations can realize huge efficiency gains and increase the chances of success.

If you're able to successfully validate each of these considerations and can build a community effort around the initiative, then automating your migration effort can be a great option.

We believe that tooling and building an iterative strategy play a huge role in streamlining the automation process.

Tooling

We believe that tooling and building an iterative strategy play a huge role in streamlining the automation process. In our engagement with a large financial services organization, we leveraged tools such as **Spring Boot® Migrator** and **Cloud Suitability Analyzer** to automate the middleware migration process. Spring Boot Migrator helps us analyze the source code of existing applications and create new source code by applying automated recipes to the results. So it's much easier to move between technologies—even when the technology is referenced in source code. On the other hand, Cloud Suitability Analyzer allows us to analyze a customer's application estate using custom rules by looking for patterns in the source code.

Spring Boot Migrator

Spring Boot Migrator uses abstract syntax trees (ASTs) to find underlying patterns within code and map these to automated recipes that can perform tasks, such as migrating from a legacy framework to Spring or upgrading from one version to another.

Figure 1: A visual of how Spring Boot Migrator works.

Spring Boot Migrator has an interactive shell but can also be run in a non-interactive manner if users need to make bulk changes to a range of applications or integrate the tool inside a pipeline.

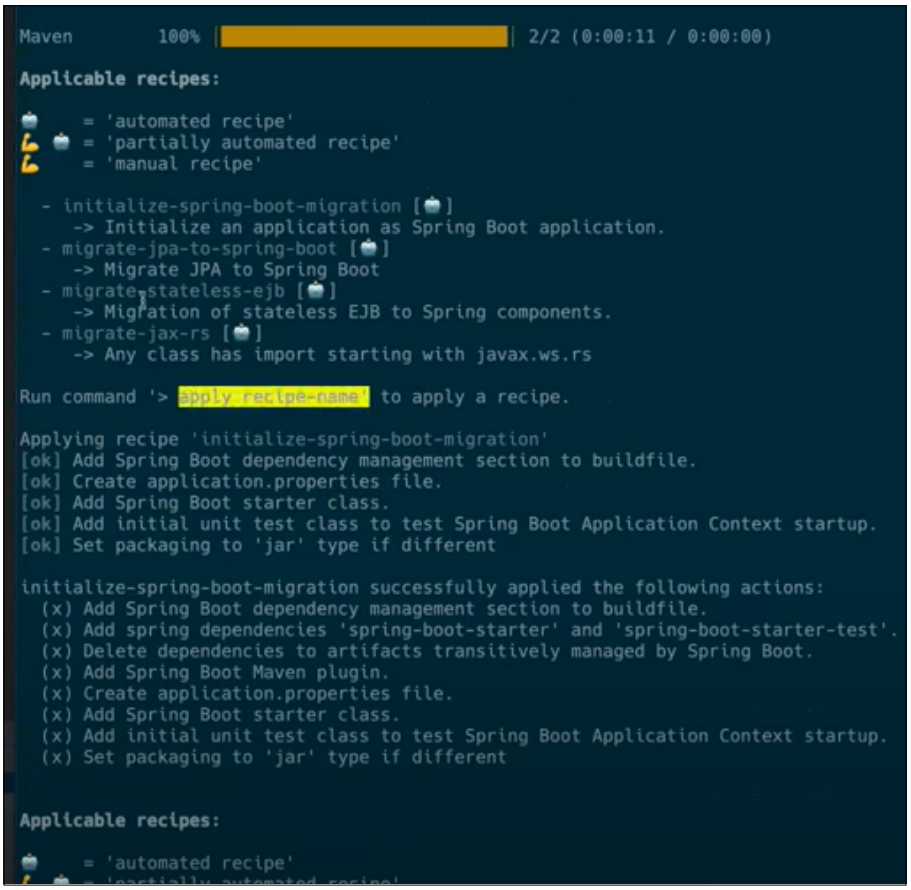
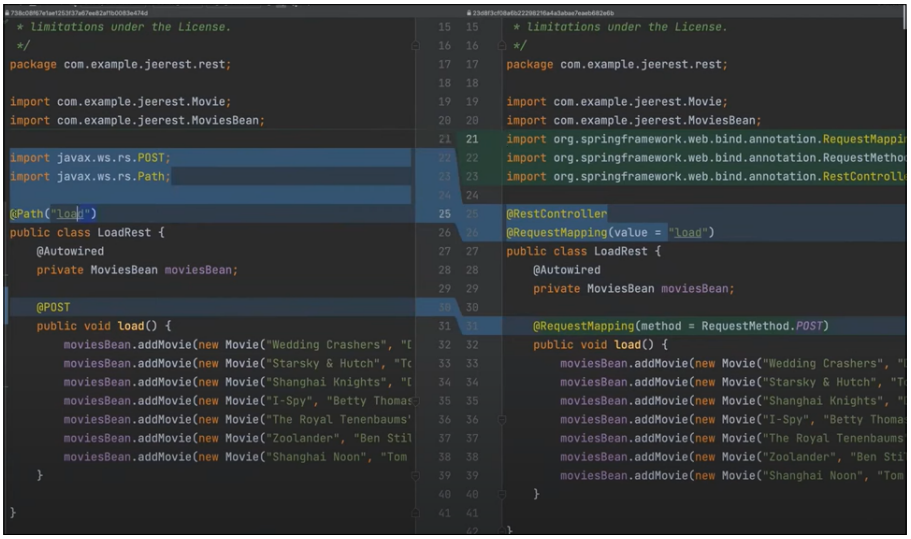


Figure 2: Users can choose which recipes to apply to suit their needs via an interactive shell.

When a user selects a particular recipe, Spring Boot Migrator runs a sequence of actions against the application source code to achieve the desired result.

In the example shown in Figure 3, Spring Boot Migrator detects the need to change the syntax of a POST rest endpoint through annotations and import the relevant functions into the code. This action was triggered by the Spring Boot Migrator recipe detecting a legacy method of declaring endpoints.



```
 * limitations under the License.
 */
package com.example.jeerest.rest;

import com.example.jeerest.Movie;
import com.example.jeerest.MoviesBean;

import javax.ws.rs.POST;
import javax.ws.rs.Path;

@Path("load")
public class LoadRest {
    @Autowired
    private MoviesBean moviesBean;

    @POST
    public void load() {
        moviesBean.addMovie(new Movie("Wedding Crashers", "D
        moviesBean.addMovie(new Movie("Starsky & Hutch", "To
        moviesBean.addMovie(new Movie("Shanghai Knights", "D
        moviesBean.addMovie(new Movie("I-Spy", "Betty Thomas
        moviesBean.addMovie(new Movie("The Royal Tenenbaums'
        moviesBean.addMovie(new Movie("Zoolander", "Ben Stil
        moviesBean.addMovie(new Movie("Shanghai Noon", "Tom
    }
}
```

```
 * limitations under the License.
 */
package com.example.jeerest.rest;

import com.example.jeerest.Movie;
import com.example.jeerest.MoviesBean;
import org.springframework.web.bind.annotation.RequestMappi
import org.springframework.web.bind.annotation.RequestMetho
import org.springframework.web.bind.annotation.RestControll

@RestController
@RequestMapping(value = "load")
public class LoadRest {
    @Autowired
    private MoviesBean moviesBean;

    @RequestMapping(method = RequestMethod.POST)
    public void load() {
        moviesBean.addMovie(new Movie("Wedding Crashers", "
        moviesBean.addMovie(new Movie("Starsky & Hutch", "T
        moviesBean.addMovie(new Movie("Shanghai Knights", "
        moviesBean.addMovie(new Movie("I-Spy", "Betty Thoma
        moviesBean.addMovie(new Movie("The Royal Tenenbaums
        moviesBean.addMovie(new Movie("Zoolander", "Ben Sti
        moviesBean.addMovie(new Movie("Shanghai Noon", "Tom
    }
}
```

Figure 3: A before and after of a simple transformation performed by Spring Boot Migrator.

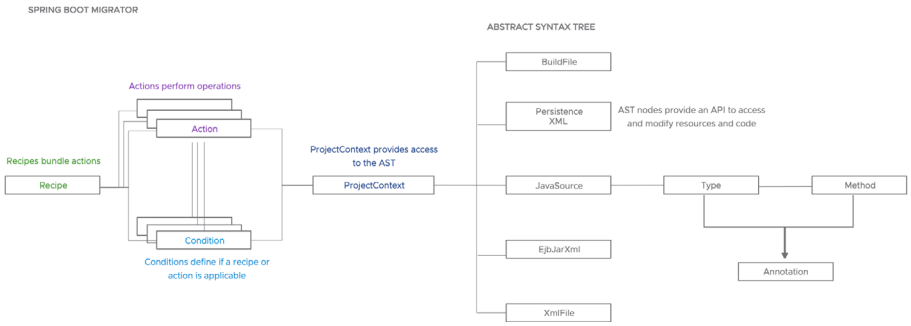


Figure 4: A demonstration of how Spring Boot Migrator generates an AST.

When Spring Boot Migrator is first run, it scans the application that the user specifies and leverages open rewrite technology to generate an AST of the application. The AST creates the structure of the code, allowing Spring Boot Migrator to analyze it quickly. Using the AST of the application, Spring Boot Migrator matches recipes that can be applied to the application, and users can select which recipes they want to apply.

Once the user applies the recipe, Spring Boot Migrator uses the AST to modify the source code by performing one or more actions defined in the recipe. By default, each change made by Spring Boot Migrator is stored in the Git commit history, so developers can see the step-by-step changes the tool makes. Recipes can be defined in either YAML format or as a Spring bean.

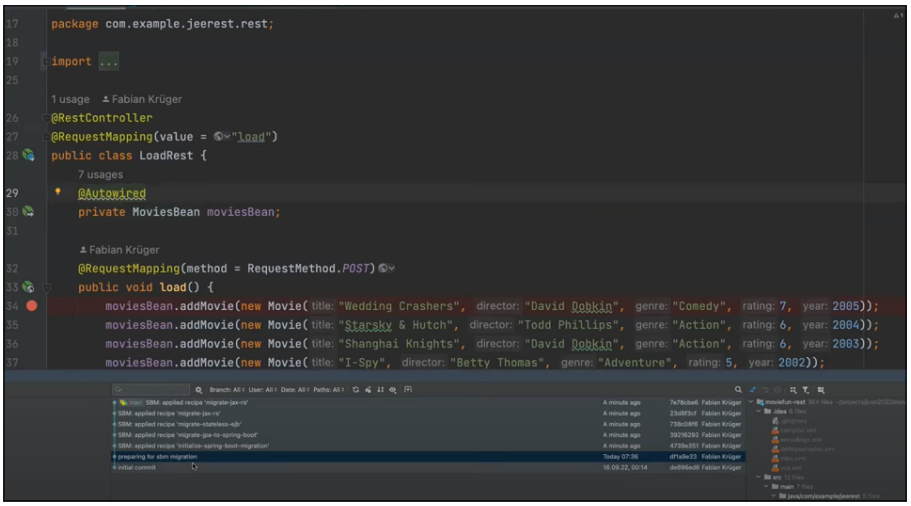


Figure 5: Git history can help developers understand the translation steps.

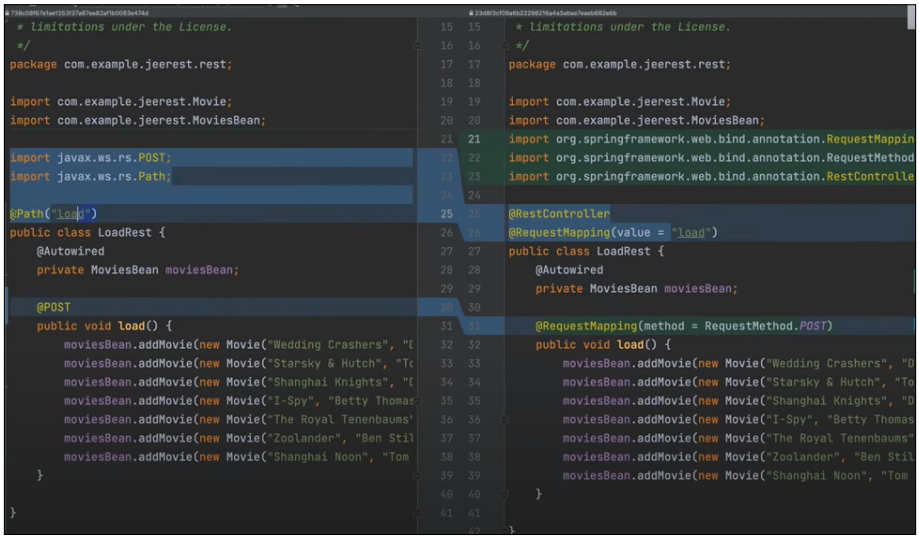


Figure 6: Using Git diff to understand the migration steps.

```
@Bean
public Recipe myRecipeDeclaration() {
  return Recipe.builder()
    .name("the-recipe-name")
    .description("Some description")
    .order(90)
    .condition(new ..SomeCondition("Some value"))
    .action(
      FirstAction
        .builder()
        .description("First action description")
        .condition(
          FileMatchingPatternExist.builder()
            .pattern("/**/ejb-jar.xml")
            .build()
        )
        .build()
      )
    .action(
      SecondAction
        .builder()
        ..
      )
    .build();
```

Figure 7: A simple template for a recipe, featuring the condition and action part of the recipe.

Ideally, recipes create new compilable application code. Through user research, we have found that developers find it frustrating to use code that has been translated and doesn't compile. When creating your own recipes, we recommend that one of the acceptance tests for the recipe logic be that it always results in compilable code.

Often, a recipe will not be able to fully translate an application and all of its logic. Therefore, it's useful to include logic that inserts the remaining untranslated code into the new application code as //todo: or //fixme comments within the code. This way, developers have a clear understanding of what remains to translate within the application.

We have found that it typically takes about one-third of the time to create a recipe as it does to translate the source code manually. As a best practice, we recommend that developers perform the manual translation of an application first, then retrospectively create Spring Boot Migrator recipes for the second translation automatically. This process helps ensure the quality of the code and

provides an opportunity for a code review by other developers. Also, having a clear idea of the before and after states helps create automated tests for the recipe creation.

Cloud Suitability Analyzer

Cloud Suitability Analyzer works by looking for certain patterns within application source code. It contains a list of rules used to rate how cloud native an application is. Individual applications, or even entire portfolios of applications, are scanned using these rules to get a rating of the difficulty to move applications to the cloud.

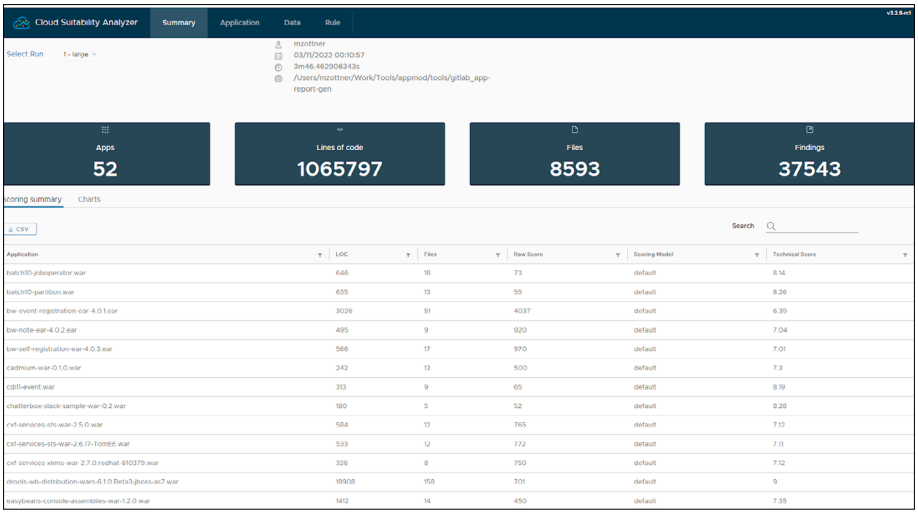


Figure 8: An example application scan of an application portfolio.

As demonstrated in Figure 8, the technical score in the final column shows how cloud native an application is likely to be on a scale of 1–10. A score of 10 indicates the application is probably very cloud native. A lower score indicates the application might need some rework before moving to a runtime environment, such as Kubernetes.

Cloud Suitability Analyzer is designed to provide insights on how modern an application portfolio is and supplement additional manual reviews of applications.

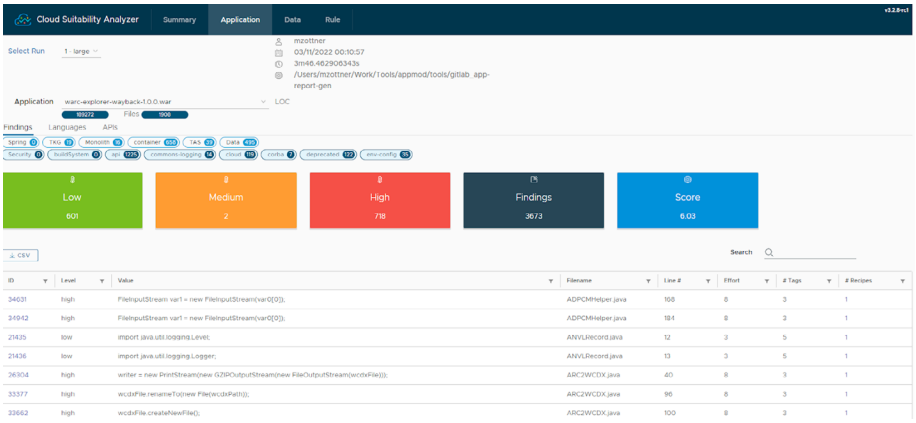


Figure 9: An example Cloud Suitability Analyzer report of potential issues with porting applications to Kubernetes.

Cloud Suitability Analyzer provides line-by-line recommendations based on a ruleset developed through years of running modernization practices for customers. It's possible to insert specific advice depending on each rule, which allows companies to share solutions to particular problems within their user base. At Tanzu Labs, we use this feature extensively to share the knowledge we've gained through hundreds of modernization engagements.

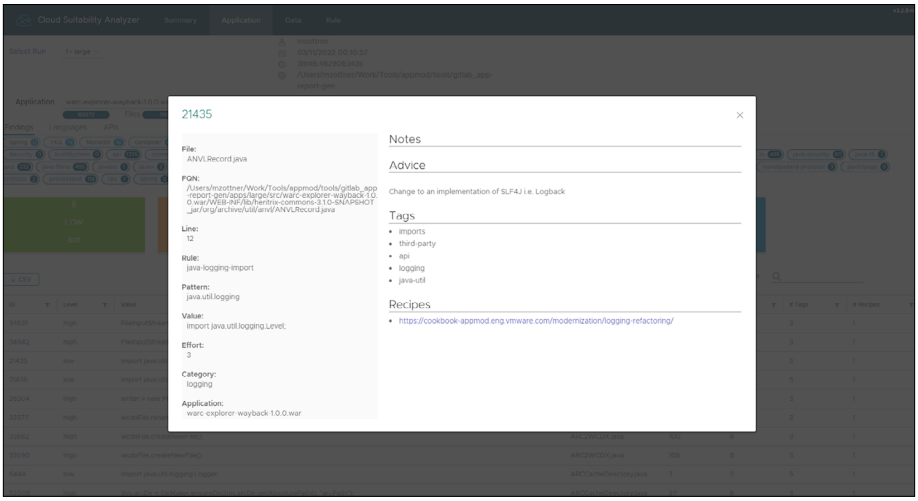


Figure 10: An example of Cloud Suitability Analyzer recommendations.

In the example shown in Figure 10, the application has been detected as using a logging library, which can cause problems when running on a cloud Kubernetes runtime. Notice the link to the recipe points to the Tanzu Labs extensive knowledge base of modernization recipes. In this case, a manual recipe provides the developer with advice on removing the offending library.

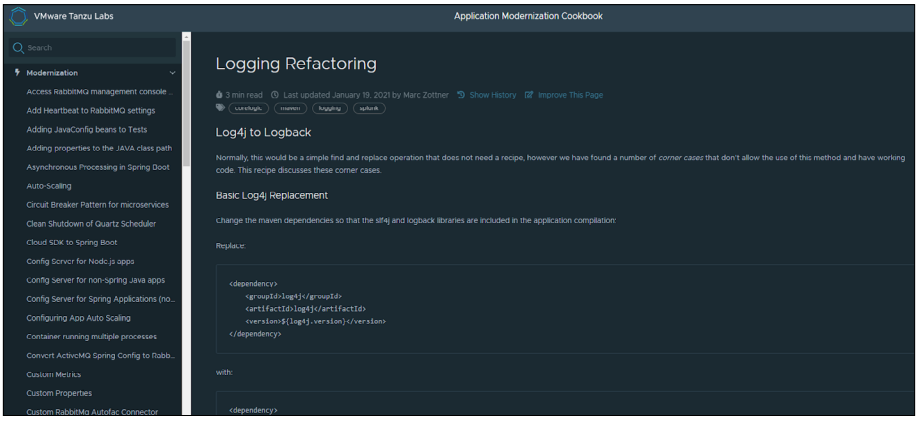


Figure 11: View the linked recipe where the VMware Tanzu® team has written instructions and advice on what to look for when changing logging mechanisms.

Cloud Suitability Analyzer comes with rules detecting anti-patterns for cloud native applications, such as applications that store state to local storage. It also uses pattern matching to detect legacy frameworks not suitable for moving to cloud native platforms, such as Kubernetes.

Cloud Suitability Analyzer can be modified by adding/modifying and removing rules into its ruleset. For example, if a company uses a custom non-cloud library for a particular piece of logic, a rule can be easily added to include to match for that particular library.

Figure 12: An example rule from Cloud Suitability Analyzer.

Rules can be captured in a range of ways. The example in Figure 12 looks for RegEx matches within files called "config" to detect the usage of Windows authentication. Cloud Suitability Analyzer has rules that detect common middleware technology, but for the financial services engagement, we wanted to get to a more granular level of detail. This project required capturing the usage of every middleware functionality to understand the usage across the application estate.

To achieve this, we needed to create a matching rule for every function the middleware used. The middleware we wanted to migrate used XML to describe its functions. XML definitions are described in the XSLT documentation, which provides information on how XML should be structured.

Using the XSLT file, we created a script that created a rule for every possible middleware feature usage. This equated to approximately 500 rules that were created for Cloud Suitability Analyzer. When performing a typical Cloud Suitability Analyzer portfolio scan; this level of granularity would be too high. In fact, it would just add noise to the rest of the results because, when analyzing cloud readiness, you typically don't need to know the exact usage of frameworks or middleware. For this reason, we decided not to add the new rules into the main Cloud Suitability Analyzer repository.

Figure 13: Using the XSLT file to create a script for middleware feature usage and generate rules for Cloud Suitability Analyzer.

To discover the usage amounts within the application estate, we used data from the rules page. This provided a count of all the matches across the application estate.

Cloud Suitability Analyzer stores all of its results into a directly connected database, so we used this functionality to analyze the results within a Jupyter Notebook for data analysis. This allowed us to experiment with different views of the data and choose the most suitable application for initial modernization.

How to automate migration

Now that we've established the kind of tooling needed to start a successful middleware transformation initiative, let's establish the how of executing.

In our engagements with customers, we built a multiphase working practice based on tooling and iteration to achieve results quickly. This section explores a sample of the strategy we built for the financial services organization to quickly migrate to open source frameworks.

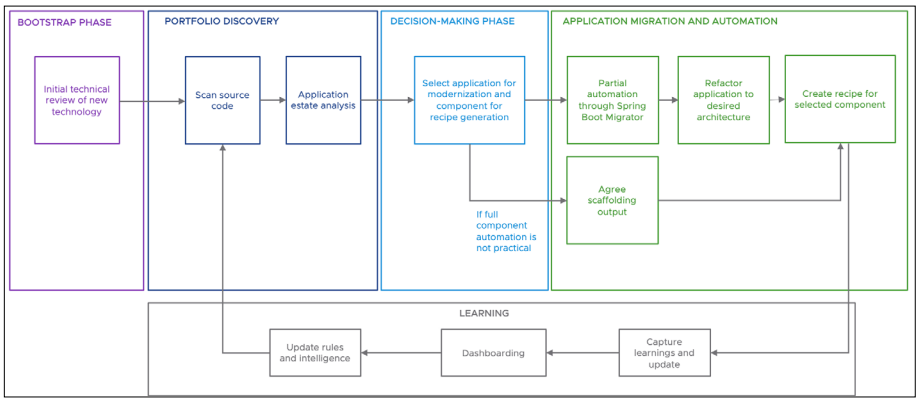


Figure 14: Our working practice.

1. Perform an initial review of new technology

In the beginning stage, also known as the bootstrap phase, we modified our tooling to better capture the patterns within the enterprise service bus. The ESB stored all its logic in a proprietary XML format, and we wanted to use our existing tooling (i.e., Cloud Suitability Analyzer) to analyze the estate. Cloud Suitability Analyzer is primarily used to provide guidance on how cloud native applications are, but we needed to use it to provide a count of all the functions that our customer's applications used (through XML components). It's easy to add custom rules into Cloud Suitability Analyzer. We used the XSLT documents (which describe how the XML files were formed) to create new rules to suit our needs. So, in the first two days, we built a simple website to create custom Cloud Suitability Analyzer rules for every possible XML component, and generated more than 200 custom rules to get the necessary level of granularity for the project.

2. Use automated tooling to scan the customer estate

With the new rules we built, we scanned our customer's entire portfolio to get an in-depth profile of which ESB features were most frequently used and categorize them with XML tags. We wanted to see the usage of these components as well as the patterns between components within the application estate (also known as cluster archetypes).

3. Find the best application to migrate

From there, we were able to select the application we wanted to migrate first. This application would ideally fit the following criteria:

- Have relatively low complexity
- Use the most frequently used components
- Have a developer who can validate the migration

A key principle of the Tanzu Labs modernization approach is to get started quickly to learn by doing, so we wanted the application to be relatively small, but we also wanted a popular archetype to create recipes for multiple components from a single application.

But most importantly, we wanted to ensure that we would be able to validate the quality of the source code being generated by Spring Boot Migrator. It was critical that our customer's developers peer review any output from the tool, so we made sure to select an application where the development owner was available to review the resulting code.

4. Migrate application and create recipes

Once we had selected the application that fit our criteria, we quickly kick-started the modernization effort and manually migrated our first application. We knew it was important to migrate the application before creating the recipes to validate with the developer representative that the new application worked as expected. Once we received the go-ahead from the developer representative, we then created recipes for the selected components for automation. The new application became our test criteria. The hope was that if we ran the old application code through Spring Boot Migrator, then the resulting code would be as close as possible to the new application.

5. Introduce learning and capture

We then captured our learnings, improved our tooling, and started the process again. The improvements in the tooling were made by adding extra rules within Cloud Suitability Analyzer to capture more detail within applications and pointing to the newly created automated recipes.

We updated Cloud Suitability Analyzer with details of the Spring Boot Migrator automation and shared it to the wider team. Teams from across the company could scan their code with Cloud Suitability Analyzer and check how much coverage Spring Boot Migrator currently had for their particular application.

During the six-week engagement, we were able to iterate over several applications and develop recipes covering approximately half of the application estate.

Finding ways to optimize your own migration strategy

We believe that leveraging source code repositories and staying aligned as a broader team plays a critical role in optimizing a middleware migration strategy. By using source code repositories as the source of truth for an application's migration status, it's possible to track the progress of an application migration effort—even before it's moved to the target infrastructure. A few characteristics of an optimized migration strategy include:

- Sharing learnings across teams and building a centralized set of tooling (Spring Boot Migrator and recipes)
- Making strategic choices on what applications to tackle next
- Gaining visibility of total progress of the migration effort

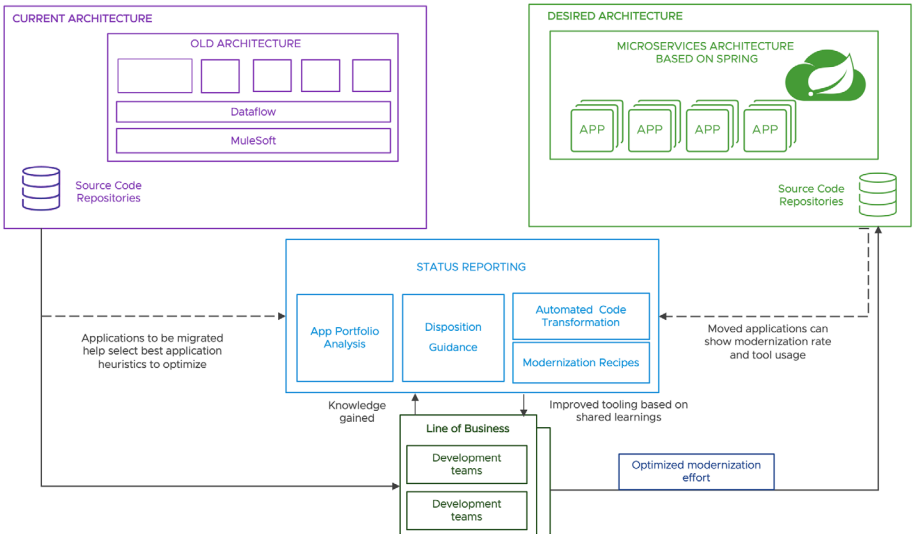


Figure 15: Ensuring alignment with status reporting and shared learnings.

Outcomes

By building an iterative migration strategy and leveraging sophisticated tooling for our financial services customer, we were able to produce high-quality source code, reduce developer toil, and demonstrate significant ROI.

Improved quality

One of the biggest outcomes from our financial services engagement was developing high-quality, reliable source code. To get universal buy-in for a new tool, we've found the tool must be trusted, and we wanted to ensure that each recipe and the output was validated by developer peers at every step of the way.

During the engagement, we discovered a few positive side effects of using automated source code generation. These were:

- We can embed best practices into the recipes we created to generate code for the new application and create unit testing boilerplate code
- The generated code helped developers who were unfamiliar with the newer frameworks learn about their workings
- Standardization between applications helped with developer onboarding

Having a familiar way of developing code meant developers can more easily understand other applications that have been modernized. Developers knew the general shape and structure of applications because the same format was used across the estate.

Early on within the project, we got feedback from some developers on our choice of technology. We decided to use Spring Integration as the framework due to it following the enterprise integration pattern model, which was the same model that our ESB followed. Many of the developers hadn't used Spring Integration before and were cautious about adopting and having to learn it. A nice side effect of having code generated automatically is that developers can use the generated code to reduce the learning curve of the new framework.

The recipes are open source and can be modified to individual needs if required. If a developer wanted to take a different approach, they can by simply reviewing the recipes and making the relevant changes themselves.

Reduction in developer toil

Another major win during the engagement was that we significantly reduced developer toil. To test whether Spring Boot Migrator actually succeeded at reducing developer toil, we asked the customer to choose a reference application of their own to demonstrate the results. They selected an application they described as being a relatively complex, non-trivial example. Spring Boot Migrator was able to fully generate 75 percent of the source code, and partially migrate about 10 percent.

During the partial migration, Spring Boot Migrator places suggestions for code conversion in place, and the developer has a choice of which approach to take depending on certain contexts (e.g., the type of database they would like to connect to).

The remaining application code had to be generated manually, but as the unconverted code was inserted in place as application TODO comments, it was agreed that the final effort was significantly easier. As a result, the customer's reference application was fully converted to Spring in just a couple of hours, which had previously taken development teams days.

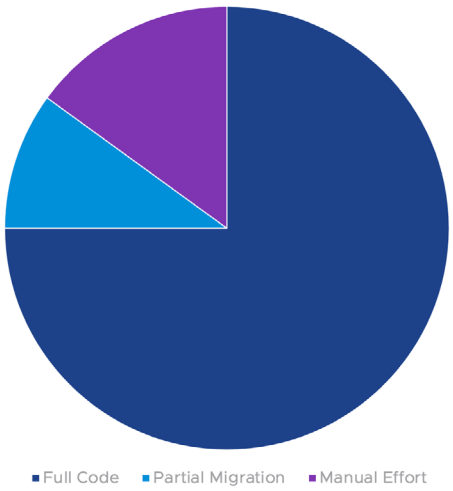


Figure 16: The reference application code conversion.

Proving the ROI of automation

Another outcome of the engagement was that we were able to demonstrate clear ROI of automation. Using Spring Boot Migrator can make developers' lives easier, but we were also able to prove that the effort involved in creating the automation recipes outweighed the benefits of writing the applications the old-fashioned way.

We found that creating a recipe for converting each component typically took about one-third of extra time. This extra effort varied a bit as some components didn't map as easily to Spring functionality, but it tended to hover around one-third. Given the massive decrease in code that developers needed to write on the converted applications, the cost-benefit trade-off was extremely high.

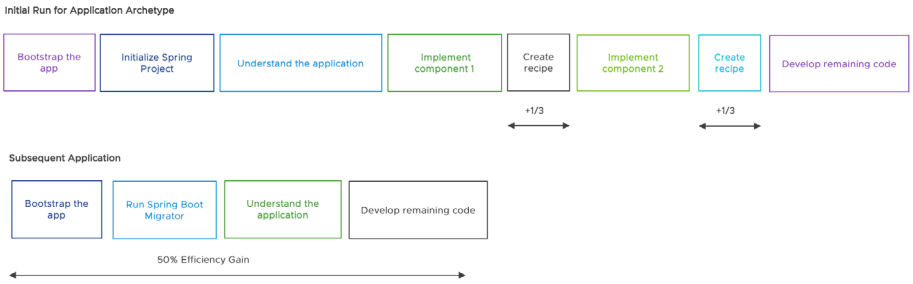


Figure 17: A visual of the process and takeaways from automating middleware migration using Spring Boot Migrator.

The key takeaways from the process shown in Figure 17 are:

- Generated code helps developers understand the app structure and provides manual guidance.
- Recipes are a fully open source, code-to-code solution, free to adapt to your preferences.
- Created reusable recipes add about one-third of extra time (for one time only).
- For the reference application, 75 percent of the final code was automatically generated by Spring Boot Migrator.
- Spring Boot Migrator includes update and migration recipes.

But to fully ensure the ROI of generating recipes, it's important to understand the big picture of the application estate. Our customer used certain components far more than others. So by choosing the most frequently used components, we were able to demonstrate that the effort of creating automated recipes far outweighed the benefits of the source code that was generated.

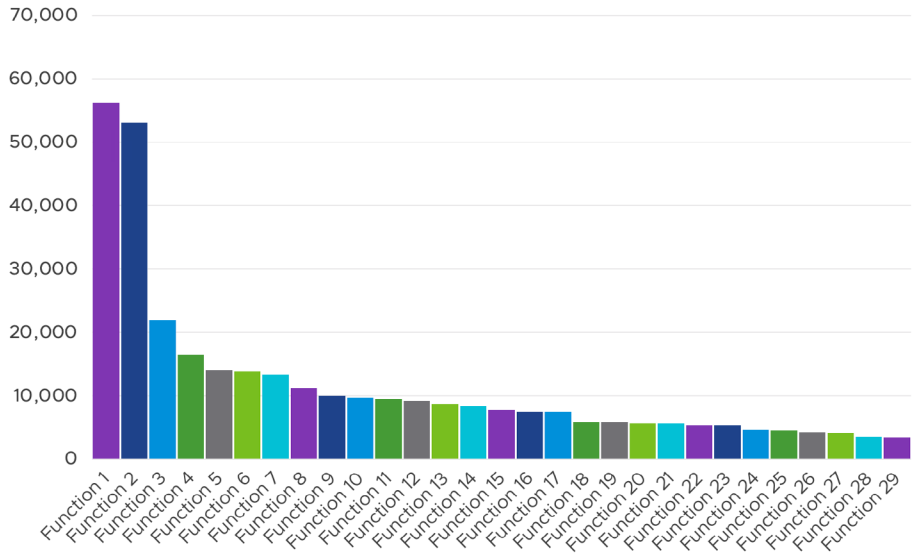


Figure 18: Most applications fit into a few common archetypes.

We also recognized that there would eventually come a time where it would no longer make sense to automate the remaining components because the coverage that we could provide would diminish as we moved to less-used components within the application estate, or components that didn't map directly to Spring functionality.

But overall, automated source code creation has proven to be a clear winner in the next generation of migration tools, with the potential to completely change the way companies modernize their applications. The benefits of reducing the mundane, repetitive tasks associated with application modernization for developers and freeing them up to work on high-value work are huge.

Conclusion

As the competitive landscape continues to evolve, we know that leveraging cloud native frameworks has become more critical than ever. Although automating the migration process from legacy middleware to open source frameworks such as Spring can feel like a huge undertaking, we believe that with the right tooling and strategy, it can be done efficiently and prove immediate ROI. In our years of work helping enterprises move to open source frameworks, we've seen how effectively leveraging recipes for automation helps organizations migrate much more efficiently. For support in building out your own middleware migration strategy, contact your VMware account team.

Learn more

To get started, reach out to your VMware Tanzu sales representative to schedule a meeting.

For more information or to purchase VMware products

Call 877-4-VMWARE (outside North America, +1-650-427-5000), visit vmware.com/products, or search online for an authorized reseller.

Resources

Tooling

Spring Boot Migrator

Cloud Suitability Analyzer

- Blog post: Cloud Suitability Analyzer: Scan and Score Your Apps' Cloud Readiness for Faster Migration
- Video: How Cloud Suitability Analyzer Can Help Speed Up App Modernization
- GitHub: Cloud Suitability Analyzer

Item No: Automate Middleware Migration to Open Source_VWright 4/23