ToLambda--Automatic Path to Serverless Architectures

Conference Paper · May 2019

DOI: 10.1109/IWoR.2019.00008

CITATIONS
7

READS
723

1 author:

Alex Kaplunovich
University of Maryland, Baltimore County
10 PUBLICATIONS **70** CITATIONS

SEE PROFILE

ToLambda–Automatic Path to Serverless Architectures

Alex Kaplunovich
Department of Computer Science
University of Maryland
Baltimore, MD 21250, USA
akaplun1@umbc.edu

***Abstract*—Serverless architectures are becoming computing standard and best practice. It is inevitable that more and more software systems will embrace the trend. Our tool toLambda provides automatic conversion of Java monolith application code into AWS Lambda Node.js microservices. During the refactoring, we provide assorted useful transformations of the original code and generate all the necessary artifacts to deploy the generated functions to the Cloud. In this paper we will describe the challenges we have faced including parsing, transformation, performance and testing. We will also underline the advantages of serverless compared to other architectures. Our approach will help to migrate systems to serverless microservices easier and faster.**

Keywords—Serverless, Microservices, AWS, Cloud, Lambda, Automation, Refactoring, Transformation, FaaS, Parsing, Java, JavaScript, Node.js

I. INTRODUCTION

Serverless architectures refer to application functions that run in ephemeral containers (Function as a Service or "FaaS") [1], [11]. They allow developers to concentrate on their granular code (called function or microservice). These functions can be deployed and tested independently. The cloud provider will be scaling microservices automatically as needed (up and down). The developers should not worry about servers and their configuration. System owner should not buy or provision any hardware (in house or in the Cloud). Users save money because they are paying only for compute time used. They do not pay for idle server time saving up to ninety percent over a cloud VM.

Serverless is becoming a modern computing standard despite of its young age (Lambda service was introduced by Amazon in 2014). As [13] demonstrates, many companies are moving to Serverless Cloud Technology. Currently, all the leading Cloud provides offer FaaS.

The driving force and motivation for this work was the intention to migrate an existing mobile monolith application to the best Cloud architecture. Our goal was to automate the process as much as possible to improve scalability, reliability and architecture.

The leading Function as a Service (Faas) Cloud Providers, prices and supported languages can be found in Tab. I. We can see that all the leading providers (Amazon, Microsoft, Google and IBM) support microservices and invest into new features development and popularization of event based cloud functions. Modern Cloud conferences and Serverless conferences present the best practices and state of the art serverless solutions. More and more software development companies (large and small) are using cloud functions and benefit from them. Nowadays, cloud functions (Lambda micro services) – the fastest growing Cloud service revolutionizing the industry.

Many venture capitalists are investing into Serverless companies, products and tools [10] and thinking that it is the future of software development. It is a very popular trend, or hype, and more and more people are involved into Serverless conferences and architectures.

TABLE I. SERVERLESS FUNCTIONS COST AND LANGUAGES

Cloud Provider	*Service Name*	*Languages Supported*	*Price $*
Amazon	Lambda	Java Node.js Python C# Go Ruby	.20 per million calls
Microsoft	Azure Functions	Node.js Python C#	.20 per million calls
Google	Cloud Functions	Node.js Python Go	.40 per million calls
IBM	Cloud Functions	Java Node.js Python	0.000017 per second, per GB

Serverless architectures significantly simplify software development since there is no need to worry about servers (infrastructure), their configuration, up time and scalability. As you can see, functions can save a lot of money because 20 cents per 1,000,000 invocations cannot be compared to any server procurement (even in the cloud).

We have been using AWS Lambda because it was released first (November, 2014 vs. Microsoft's November 2016) and Lambda has more features than any other FaaS offering.

As Peter Sbarski said in [2] "Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. This exciting new shift in architecture will grow quickly as software developers embrace compute

services such as AWS Lambda. And, in many cases, serverless applications will be cheaper to run and faster to implement".

II. RATIONAL

A. toLambda Background

The pace of Cloud technologies is extremely fast. The best practices are getting outdated within a year or two. We started from a monolith mobile application on the AWS Cloud using ElasticBeanstalk, CloudFront, S3 and RDS database several years ago. When serverless was introduced in 2014 - we have realized how beneficial it was. We decided to convert our application into serverless microservices.

Manual conversion would have been too expensive and time consuming. We have decided to apply automation and create a tool that converts our application into Lambda automatically.

B. Java vs. JavaScript

For years computer scientists were discussing compiled vs. interpreted languages. Both are Turing complete. Most of us were taught that compiled strongly typed languages perform better. The reasons are – advanced compiler optimizations, compile time validations and usage of native code. There is an ongoing discussion about strongly typed compiled programming languages (Java, C#, Go) versus interpreted languages (JavaScript, Python).

Interpreted languages have their own advantages – easier to implement, less verbose, smaller code artifacts, and no compilation overhead.

There were numerous benchmarking efforts [7], [8] and [9]. Most of the efforts have proven that interpreted Node.js outperforms compiled Java when used for Lambda functions.

Currently, AWS Lambda supports the following programming languages – JavaScript, Python, Java, C# and Go. Every aspiring computer scientist needs to choose a language to write their microservices. The main criteria to consider – performance and time to implement the application.

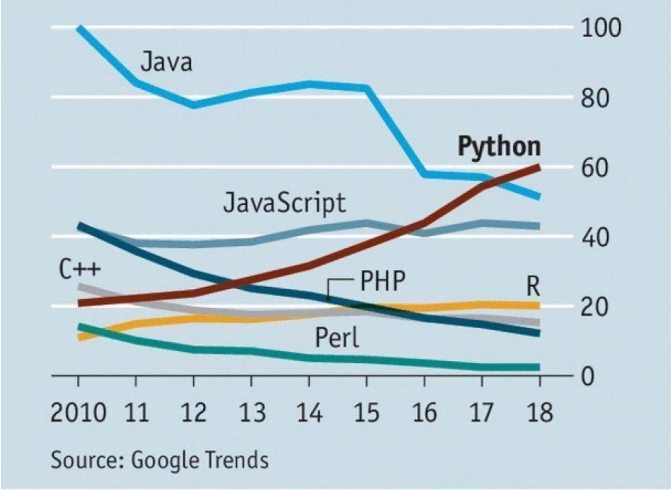

Fig. 1. Programming languages search popularity

As Fig. 1 (generated from Google Trends) shows, the number of searches for Java drops drastically. Analyzing the programming languages evolution and performance, we have realized that it will be beneficial to replace our original compiled Java program with less verbose, well performing, interpreted Node.js (JavaScript) code. After we have completed the conversion, we have realized that the tool can be used for a wide range of generic applications, converting them to a bunch of microservices automatically. Many other organizations can benefit from automatic conversion to Lambda microservices and code transformations implemented during the conversion process. Moreover, we have realized that it is not extremely difficult to add more customized transformations to your code base during the conversion.

C. Monolith vs. Microservices vs. Serverless

For years monolith applications were developed and deployed into numerous servers. As Fig. 2 demonstrates, we deploy all the application methods to every server. If, for any reason, one method is heavily used, we will have to deploy the whole monolith application into containers, using a lot of resources including servers, memory, disk, etc.

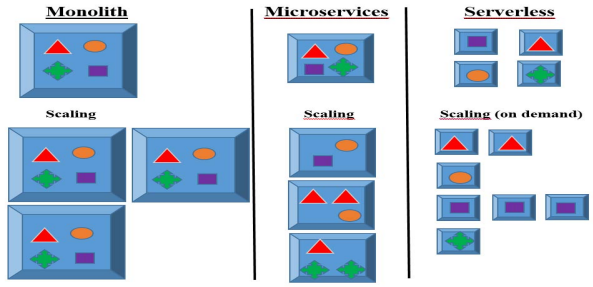

Fig. 2. Monolith vs. Microservices vs. Serverless

If one method is heavily used, we cannot deploy (replicate) that method for scaling without the rest of the monolith application methods. Modern Cloud Architectures promote granular scaling that improve services scalability and availability, leading to resilient systems.

Microservices - a relatively new trend in software development. Francesco et al [5] tries to classify and generalize the different approaches and best practices. Many industry leaders (including Netflix and Amazon) have embraced the concept and provided reliable and efficient solutions and implementation frameworks.

Serverless Architectures support one method deployment and automatic (on demand) scaling (up and down). Depending on the load, the methods are deployed into the appropriate containers (servers). Such an architecture provides a lot of benefits including cost and resources savings, granularity, efficient software design and better fault tolerance.

D. AWS Lambda

AWS Lambda service was introduced in 2014. It is a natural extension to the microservices concept augmented by the Cloud ecosystem integration and automation.

Lambda function is code (in Python, JavaScript, Java, C# or Go) deployed to the Cloud. The code can be triggered either by a direct call (API or URL), or by configured events. Almost any cloud event can be configured to trigger Lambda – database row updated, file removed, Alexa request, CloudFormation stack, CloudWatch logs, and many more. Reference [6] contains more details about event sources which can trigger Lambda.

Event based invocation and auto-scaling provide limitless possibilities and make cloud function so popular. Developers should not worry about provisioning, discovery or managing servers. It will be done automatically. If one or more function is triggered very often, the Cloud ecosystem will create all the necessary infrastructure automatically to scale your application. The functions can also be grouped and versioned.

In conjunction with API Getaway and Security services, Lambda allows to create state of the art highly secure, available and scalable applications.

Fig. 3. Demonstrates Lambda function triggering diagram. The function is triggered by Kinesis and writes data into DynamoDB.

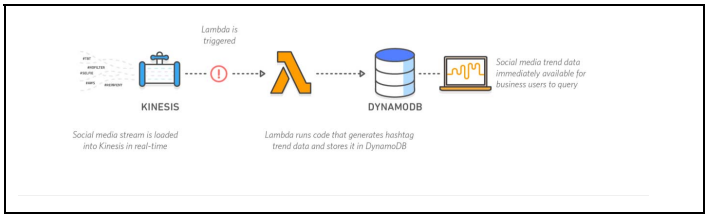

Fig. 3. Lambda function triggering diagram

E. Node.js conversion Advantages

toLambda provides the following advantages. Since Node.js (JavaScript) is an interpreted language, there is no necessity to compile your code before deploying it into the Cloud. It saves substantial amount of time and makes is much easier to deploy your whole application or a single cloud function.

Node.js is less verbose than Java. The amount of code decreases because identical operations require much less characters in Node.js than in Java. For example, variable or array declarations, numerous modifiers, initialization, collection manipulations, getter and setter calls.

Node.js is widely used for web development, it is a primary language for writing Alexa skills and it is very well integrated with Amazon libraries. You easily can access any AWS service from Node.js lambda function using AWS SDK for JavaScript.

JavaScript promises makes asynchronous development much easier and structured. Especially with the firebird library, we do not have to deal with confusing waterfall and callbacks for asynchronous calls. Promises help Node.js functions to be readable, reliable and easily handled.

III. TOLAMBDA IMPLEMENTATION

Since the original application has been written in Java, our tool is implemented in Java as well. We have been using a JDT parser to parse the source code of the monolith application. We are using the event-based SAX parser and process events on the fly, generating the resulting code and applying desired transformations.

The input to our tool contains a compressed directory of Java source files and a list of service classes. Those classes will be extracted and for each public method, we will be generating a corresponding Lambda function code in Node.js.

Each of those functions will contain all the constants and methods referenced in the original code. In other words, we decompose a monolith application into self-sufficient microservices containing all the necessary code to execute.

We also do several code transformations during our parsing and code generation. Examples:

- System.out.print(ln) -> console.log
- Logger output -> console. + corresponding method
- Logger classes creation and instantiation -> skip
- Annotations -> comments
- Class cast -> comment
- Getters and setters -> .propertyName (see below)

Instance1.getValue1() becomes -> Instance1.value1
Instance2.setValue2(123) -> Instance2.value2=123

- Multiple class/method combinations to one method

org.apache.logging.log4j.Logger.debug(...)

ava.util.logging.Logger.debug(...)-> console.debug(...)

Since we are parsing the source code, we have unlimited power to transform one class/package/method to anything we desire.

The output of toLambda tool is very flexible. For each service class we create a directory/bucket containing all the public methods with all the dependencies, comments, constants and enumerations. Each file can be deployed as a Lambda function.

A. Architecture

Our tool is a generic automation and transformation tool. It takes a zip file with your Java monolith application source code and generates Node.js AWS Lambda functions. These methods can be deployed into AWS manually or through SAM Cloud Formation templates. Decomposition and dependency search is done completely automatically. The code generated is self-sufficient and contains all the referenced methods.

The parser processes the source code tree and generates all the necessary JavaScript code for the provided service classes. It decomposes the service classes by converting each public method to lambda function. Each generated method contains all the constant definition and the transformed code for all the referenced methods.

Fig. 4 demonstrates the workflow from the zipped monolith application file to the decomposed Lambda microservices.

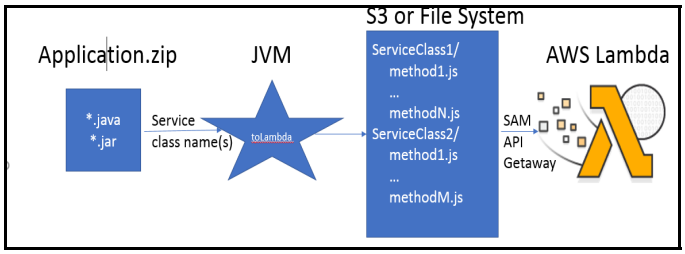

Fig. 4. toLambda Architecture

B. Output

The tool outputs AWS lambda compatible Node.js (JavaScript) files. Each file corresponds to one public method in the service class and contains all the dependencies that could be found in the source zipped file. For example, if our service method calls Class1.method2(param3, param3), the source code of method2 will be a part of the code generated. It should be easy to take those files and deploy them to the cloud either manually or using the SAM framework – Serverless Application Model [3].

If several Java methods in the class have the same name (and different parameters – Java polymorphism support), we will be generating the same number of methods with unique suffixes added to their names to identify the correct method in JavaScript (polymorphism support).

It is important to mention that we provide a list of service class names to the tool. Usually, service classes implement the business logic and called from the frontend via Ajax calls. Lambda microservices will be generated only for the public methods of the specified classes. Our approach limits the number of functions generated and deployed into the Cloud.

C. Challenges

Decomposing and Transforming a monolith application is not an easy task. We had to discover some Java features we have not thought about. To have made a successful implementation dealing with

- Polymorphism
- Constructor hierarchies
- Enumerations
- Local classes and interfaces
- Anonymous classes
- Synchronized, super, this, lambda expressions
- Complex types
- Literals and constants
- Exception handling
- Loops and branching
- Asserts
- Declaration keywords
- Generics

Each of our generated cloud function (Lambda) contains an absolute minimum of code necessary to execute. To accomplish that, we must traverse the source tree multiple times, finding the dependencies and transforming all the methods called into JavaScript.

We should also discover and output all the constants and enumerations referenced in the methods exported. To avoid disambiguates, we will be prepending a full class path to the constants.

D. Decomposition

Monolith application might contain thousands of classes. In the parameter serviceClass we specify the list of classes to convert to Lambda functions. Such a choice helps the tool to identify code to start from and avoids searching in the forest of code in the dark. We will decompose into Lambdas only few service classes (not all the monolith classes), provided by the user as input.

For each of the supplied service classes, we will identify public methods and generate JavaScript code for each of those methods. Each method will contain Lambda function declarations, all the referenced methods code, constants and enumerations.

We should mention that decomposition involves recursively traversing all the dependent classes, that's why we must keep track of the methods and constants already included.

Our experience shows that the performance of our algorithm is acceptable, and we can generate lambdas from very complicated and lengthy code within seconds.

Technically, our tool might be used to "transform" any public method of any class (it should not necessarily be a service). Developers might need that to find all the dependencies of a particular method, see complexities or output file size. Sometimes it is eye opening to see which libraries, classes and methods your simple function references. It definitely will help to see what is involved to convert a certain method to a microservice.

E. Java to JavaScript

JavaScript increases its share in software development languages. Although Java is a very popular language, developers are looking for viable alternatives due to time consuming build, compilation and Java excessive verbosity. Moreover, a lot of powerful JavaScript libraries have been introduced lately in the areas of both frontend and backend, cloud and database integration. Modern developers love JavaScript and use it for many state- of-the-art systems.

Recently, Oracle started charging for Java. Given, it was a free language for decades, many organizations and faithful developers will stop using it in the near future.

1) Parsing

To translate from one language to another, we need to understand the source to transform from. Current version of toLambda is using an open source SAX Java parser (JDT). Such a parser allows us to traverse source code processing events of interest and generating all the necessary code on the

fly. Fortunately, we do not have to parse manually, because modern parsers trace many important language events – just to name a few – method declaration, method invocation, array creation, variable declaration. We use visitor pattern to process individual parsing events when they start (method *visit*) and finish (method *endVisit*). Moreover, our AST parser can distinguish identical methods, and can handle abstract classes, inheritance and polymorphism. We use these features intensively to avoid code duplication and to call (or export) the correct methods in the generated microservices.

During parsing we keep the state and keep track of all the referenced methods and constants (even from different class). After each public method parsing is complete – we recursively parse all the referenced object and add them to the generated JavaScript code.

Our parser allows us to distinguish a dynamic object from a constant, which helps us to keep track of constants and put them on top of the output files.

Collecting all the dependencies and outputting then into one deployable method is a challenge. We had to provide several parameters that help the tool to generate the correct code. The parameter SKIP_PACKAGES allows to exclude a set of packages or classes from code generation (and parameter structure display). By default, the tool will be trying to generate code for every method referenced (if it can find the code in the zipped source directory provided). The option is very useful for specifying packages or libraries we do not want to traverse further. The same parameter is used to avoid member hierarchical traversal for parameters (see below).

2) Parameters

Each Java method has a signature – a list of parameters and their types. During Lambda functions generation, we traverse a structure of each parameter and output indented member hierarchy in comments. It really helps the developers to understand the incoming parameters structure. The comments contain JSON style structure of the input parameters and the return type (output value) structure.

```
/**
 Method abstractNotimplementedMethod from class
com.ate.comlex.test.service.Service1Impl
[public java.lang.Long
abstractNotimplementedMethod(com.ate.comlex.test.domain.Test3[],
com.ate.comlex.test.domain.Test3) ]
 Method params [Test3[] person, Test3 manager]
 Input:{
   person    :  com.ate.comlex.test.domain.Test3[],
   manager      :  {
      created      :  date,
      occupation    :  string,
      gender    :  com.ate.comlex.test.domain.Gender,
      age      :  int,
      id    :  long,
      version      :  long,
      year      :  int
   }
}
Output    :  long
Method annotations @Override **/
```

Fig. 5. Generated method comment

We do output Java annotation in comments, so developers can review the generated code later and take an appropriate action if needed. Fig. 5 demonstrates generated method header with detailed original class/method name, classpath, input/output parameters (with their types), and annotations. Such a header give a comprehensive information about the source Java method and its location in the monolith code.

3) Constants

During our lambda functions code generation, we can detect all the constants used by a method (even if the constants are defined in different classes). Those constants are placed in the beginning of the function code. It is extremely helpful to see all the constants at the top of the generated code.

To distinguish the constants from different source classes we place a full classpath in front of the constant name:

```
const com_ate_util_Constants$MAX_EMBED_VAR=10;
```

F. Method Details

1) Standardizing acceptChild and acceptChildren methods

We are using an open source JDT parsed for our tool. It is an event-based parser dealing with the instances of the class *ASTNode* as a unit of parsing processing. The parser is using a visitor pattern, where for each node (ASTNode), we call an accept method that in turn calls *preVisit, accept* and *postVisit* methods.

In order to process all the node's children consistently, we have implemented a standard *acceptChild* and *acceptChildrent* methods. These methods are called whenever we want to parse and transform any Java construct into a new Lambda Serverless function.

```
final void acceptChildren(StringBuilder code, List list, String delim,
String braceOpen, String braceClose, int outputBraces,
String prepandParam) {
   if (list != null && list.size() > 0 && list.size() >= outputBraces ||
         (list == null || list.size() == 0) && outputBraces > 0)
      code.append(braceOpen);
   final boolean hasPrepandParam =
!Strings.isBlank(prepandParam);
   if (hasPrepandParam) {
      code.append(prepandParam);
   }
   if (list != null && list.size() > 0)
      if (hasPrepandParam)
         code.append(delim);
   for (Iterator<ASTNode> it = ((List<ASTNode>) list).iterator();
it.hasNext(); ) {
      ASTNode arg = it.next();
      acceptChild(arg);
      if (it.hasNext())
         code.append(delim);
   }
   if (list != null && list.size() > 0 && list.size() >= outputBraces ||
(list == null || list.size() == 0) && outputBraces > 0)
      code.append(braceClose);
}
```

Fig. 6. acceptChildren method processing a list of ASTNode(s)

acceptChild method systematically processes one ASTNode (anonymousClassDeclaration, body, expression, qualified, initializer, operator, operand, thenStatement, elseStatement, etc.) A node might contain multiple children that will be processed by the corresponding event triggered by the *child.accept* method.

acceptChildren method is called when we want to process a list of nodes (typeArguments, parameters, arguments, dimensions, expressions, etc.) In the method arguments, we specify the delimiter, brace symbols and prepend string if needed (Fig. 6). Such a method allows us to generate the output consistently. Moreover, when we will be extending the tool to other programming languages it will be pretty straightforward to change just a few methods to generate an output for a different language, change a delimiting or brace characters. The methods can be easily changed to make another standard transformation to the generated code.

2) Code generation

Since we have implemented the accept methods, we call them on every parsed Java construct (node) - (expressions, blocks, declarations, statements, etc.) We implement the methods visit and endVisit for each of the Java parsed elements. They have the logic necessary to generate necessary Javascript code for the particular node. These methods will be called during parsing. Fig. 7 has an example of the IfStatement node. We keep track of indentation and append the language specific keywords to the generated code.

```
public boolean visit(IfStatement node) {
    StringBuilder code = blockCode.peek();
    if (!(node.getParent() instanceof IfStatement))
        ParserUtil.indentBuffer(code, level);
code.append(Constants.IF_STATEMENT).append
(Constants.OPARAM);
    acceptChild(node.getExpression());

code.append(Constants.CLOSE_PARAM).append(Constants.LINE);
    final Statement thenStatement = node.getThenStatement();
    if (thenStatement != null) {
        if (!(thenStatement instanceof Block)) {
            level++;
        }
        acceptChild(thenStatement);
        if (!(thenStatement instanceof Block)) {
            level--;
        }
    }
    final Statement elseStatement = node.getElseStatement();
    if (elseStatement != null) {
        ParserUtil.indentBuffer(code, level);
       code.append(Constants.ELSE_STATEMENT).
append(Constants.SPACE);
        if (!(elseStatement instanceof Block) &&
!(elseStatement instanceof IfStatement)) {
            level++;
        }
        if (!(elseStatement instanceof IfStatement))
            code.append(Constants.NEW_LINE);
        acceptChild(elseStatement);
        if (!(elseStatement instanceof Block) &&
!(elseStatement instanceof IfStatement)) {
            level--;
        }
    }
    return false;
  }
```

Fig. 7. Processing IfStatement and calling accept methods

G. Testing

Testing serverless microservices has always been a challenge. Although lambda functions are independent, they should be working together with each other. Since the industry is very passionate about microservices, more and more tools are being created to simplify deployment and testing.

AWS SAM (Serverless Application Model) [3] and [4] provides a simple way to deploy and test functions locally or remotely. It also can generate events for triggering lambda functions. The format of SAM script is JSON or yaml. It is very similar to the Cloud Formation service templates, which are closely integrated with the whole AWS ecosystem. Fig. 8 demonstrates how to deploy and execute a Lambda function.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  MyFunction:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: index.handler
      Runtime: Node.js6.10
      CodeUri: 's3://my-bucket/myFunction.zip'
```

Fig. 8. SAM yaml script example

In December 2017, Amazon has released Cloud 9 – a full functioning Multilanguage IDE. It can edit, deploy, and export Lambda functions from AWS. It can also test and debug the microservices locally line-by-line. Such an IDE simplifies the implementation of Lambda functions, improves quality of released code and saves a lot of time.

The serverless technologies are relatively new (introduced in 2014). Many industry leaders (Amazon, Google, Microsoft, and IBM) are working hard to simplify the development and testing of event driven, serverless applications. We will see more serverless tools because Serverless microservices are becoming industry standard.

F. Heuristics

In order to implement the tool efficiently we had to review many Java features and implement several important heuristics. Given excellent performance results from Tab. II., we were able to find the necessary heuristics to optimize our tool. Given the potential source code size, we need to be able to handle large zip arches and large source files handling all the features of the original monolith application programming language.

1) Source File cacheing

The monolith application's source code is provided as a zipped archive containing a magnitude of files. During a class traversal and lambda functions generations we can potentially

visit the same files multiple times. That is why we are caching the processed Java source files in the key-value map, and during the consequent request, we just proceed with the cached source if found. Modern hash map performance is O(1) as [12] confirms.

A full source code class path serves as a key for our map. As a result, we do not need to process a zip archive to locate and extract a Java source file again. As we know, deflating archives is a lengthy operation. That is why before we start parsing, we deflate the whole archive into a local file system.

Given we have to recursively parse multiple source files again and again, we are saving substantial amounts of processing time because we do not have to use file IO operations, and use cached data instead.

2) SHA1 digest and polymorphism

Polymorphism is one of the widely used features of Java. A class can contain multiple methods with the same name, but with different parameters. To distinguish such methods, we use a *SHA1* hash. Since a digest is calculated from the full method path and signature (*IMethodBinding.getMethodDeclaration()),* it can uniquely identify a method in a class.

It helps to figure out if we need to append the method to the existing Lambda code or if the method has been already appended to the generated Lambda output.

3) Cloud Readiness

Since Lambda is a Cloud native service, we have been writing our tool targeting the Cloud deployment. With such a design and architecture, we can easily switch our tool from running locally to be deployed and hosted on the Cloud to increase accessibility and scalability.

Modular interfaces and pluggable utilities allow us to switch to the Cloud easily. Switching from FileUtil to the BucketUtil moves the tool from the file system to the cloud simple storage service (S3) buckets.

Moreover, our tool itself can be deployed as a Serverless Lambda function, allowing almost unlimited scalability and reliability.

We anticipate that most of the applications will be migrated to the Serverless Microservices Architecture in the near future. We expect the upcoming Re:Invent conference will reveal the latest Lambda and API Getaway features helping the whole industry to move to Serverless even faster.

IV. PERFORMANCE

We have been testing our tool on assorted complex Java source codes. Although we traverse the code to find all the dependencies, the performance of the tool is amazing. It takes at most one minute to process one service class generation for the most complex code we have been using (hundreds of thousands of complex files sized around 100MB).

In order to avoid infinite loops, we can configure the depth of recursive code traversal, as a parameter MAX_EMBED_LEVEL. In our experience, the value of 10 is sufficient for most of the cases.

TABLE II. EXECUTION REULTS

# Source Files	*Java Lines*	*Max embed level needed*	*# of class Methods*	*Average method generation time*
19	2714	10	100	0.92
115	17005	6	28	0.22
265	36021	7	2	1.77
1059	97447	4	22	0.21
1139	137537	5	43	0.73
1660	229396	9	57	1.39

We have tested our approach on multiple data sources (including quarter of million lines) with multiple recursive traversals to get the referenced methods. It was taking at most several seconds to generate a lambda function from a service method even with traversing source code tree ten times to extract dependencies. Tab. II demonstrates the timing and statistical results.

V. FUTURE WORK

We are transforming our application to Node.js (JavaScript) lambda functions. JavaScript is a very popular language, used widely for web frontend development and Alexa skills SDK. However, once we have completed the transformation, we have realized that it is possible to convert to other programming languages (Python, for example). Nowadays different languages have different advantages. For example, Python is the most popular language for machine learning applications. If we want to use a particular machine learning library or frameworks (TensorFlow, Torch or Keras) we can easily integrate them into our cloud functions.

Nowadays, the concept of programming language is becoming transparent. With the advances in NLP and Machine Learning, spoken languages can be automatically “translated”. We believe that programming languages will become easier and easier to translate. Finally, software development will become language agnostic with easy and automatic language translation.

Automatic code conversion from one programming language to another using RNNs (Recurrent Neural Networks) is a very promising area of Computer Science. Companies like Amazon and Google routinely use RNNs for natural language translation; and the quality of their translation is getting better is better over the years. We believe it is time to “translate” software from one language to another.

There are many new tools becoming available for faster and better Lambda development. Amazon already provides a Cloud9 IDE (Integrated Development Environment) that allows to test and deploy any Lambda function locally. Tools like that simplify the lives of software developers and let them concentrate on design, architecture and business logic.

It is also possible to “transform” any package, class or method call on the fly. For example, we can replace all file

system calls to the corresponding cloud storage methods (File.open -> S3.getObject).

Developers might also want to migrate from SQL to NoSQL databases. We can help them to automatically convert JdbcTemplate methods to the corresponding DynamoDB calls (JdbcTemplate.query ->DynamoDB.query).

Such transformations will help to migrate to cloud efficiently, utilizing the best practices, the best architectures and services available. We can also automate security constraints for the generated methods to simplify lambda functions deployment into the Cloud using API Getaway and SAM.

VI. CONCLUSION

Serverless is becoming a mainstream of computing, and the best practice of Cloud applications architecture. Our tool helps to automate the process of migrating monolith applications into microservices. It will help software industry to migrate to Cloud more efficiently, saving time and money. As we know, there are many cloud migration approaches. One of them, for example, is lift and shift – when we take the whole datacenter infrastructure and "move it" to the Cloud. Such approach turns out to be an antipattern because the organizations are inheriting all the datacenter problems with such a move.

Our system automatically generates source code, transforming the original system to better architectures, making improvements, language and structural transformations. We believe that software systems of the future will be generating code for humans. toLambda is just the beginning.

Our approach helps to create efficient and automated migration to the Cloud microservices, using the state of the art technologies, services and best practices. We are using such technologies as Lambda, API Getaway and SAM. Our tool creates modern and efficient software applications. We automate the migration to the serverless Cloud architectures saving time and money. The tool will help many software organizations to migrate to serverless architectures.

REFERENCES

[1] Martin Fowler web site. https://martinfowler.com/articles/serverless.html *(references)*

[2] Peter Sbarski. 2017. Serverless Architectures on AWS. Manning

[3] SAM Specification – Serverless Application Model, AWS, https://github.com/awslabs/serverless-application-model

[4] SAM CLI (command line interface) https://github.com/awslabs/aws-sam-cli.

[5] Paolo Di Francesco; Ivano Malavolta ; Patricia Lago Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption,

2017 IEEE International Conference on Software Architecture (ICSA).

[6] AWS, Supported Event Sources, https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html.

[7] Yun Zhi Lin, Comparing AWS Lambda performance of Node.js, Python, Java, C# and Go, https://read.acloud.guru/comparing-aws-lambda-performance-of-node-js-python-java-c-and-go-29c1163c2581.

[8] Tim Nolet, AWS Lambda Go vs. Node.js performance benchmark, https://hackernoon.com/aws-lambda-go-vs-node-js-performance-benchmark-1c8898341982

[9] Mark West, Comparing Java and Node.js on AWS Lambda, 17. sep. 2017, https://www.bouvet.no/bouvet-deler/comparing-java-and-node.js-on-aws-lambda

[10] Sarah Guo and Jerry Chen, Serverless: Changing the Face of Business Economics, May 2018,

https://news.greylock.com/serverless-changing-the-face-of-business-economics-873898f5b74a

[11] Maruti Techlabs, What is Serverless Architecture? What are its criticisms and drawbacks?, May 2017,

https://medium.com/@MarutiTech/what-is-serverless-architecture-what-are-its-criticisms-and-drawbacks-928659f9899a

[12] Tomasz Nurkiewicz, HashMap Performance Improvements in Java 8, April 2014, Performance Zone, https://dzone.com/articles/hashmap-performance

[13] Raelene Morey, "Why Companies Are Moving to Serverless Cloud Technology", September 2018, https://northstack.com/why-companies-are-moving-to-serverless-cloud-technology/