Java Code Analysis and Transformation into AWS Lambda Functions

Group Member: Baojia Zhang, Kaixuan Gao, Yuxiao Guo, Ziyu Gao

Cloud Computing and Project Design

Various Programming Models to choose from:

- Traditional Monolithic Architecture?
- Microservice?
- Function-as-a-Service (FaaS) platform?
- ……

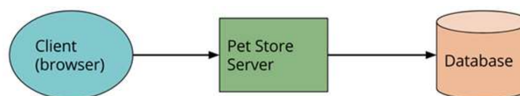
What if I want to switch to another model?

Serverless Computing: FaaS

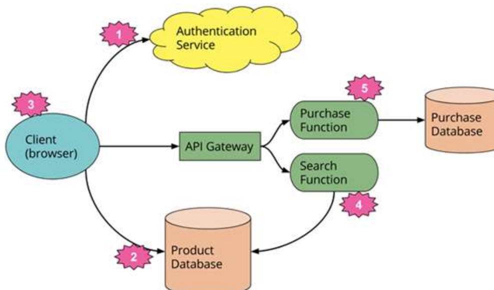
- Easy Deployment
- Pay-as-you-go
- Scalability: server capacity is automatically provisioned
- Stateless and Event-Processing System:
  - Suit for event-driven and flow-like processing patterns
- ……

Traditional Server Vs. Serverless Architecture?

Traditional Server-side Architecture



Serverless Architecture



Serverless Architecture, Mike Robert, https://martinfowler.com/articles/serverless.html

Podilizer

- Java and AWS Lambda
- Transformation from monolithic Java code to AWS Lambda units
- Let developer make policy choice at a late point in time
- Hide the actual mechanism to enact the policy

Approach

- Identified two research question
- Explained general decisions which must be taken by any transformation tool related to the programming model, state handling and the process design
- Presented both the design and the implementation of Podilizer

Research Questions

- RQ1: Is it economically viable to run a Java application entirely on FaaS?
  - Comparison baseline: a) PaaS b) IaaS
- RQ2: Is it technically feasible to automate this process? If so, what is the percentage, performance and which code is easier, harder or impossible to convert?

Programming → Execution Model Mapping

First challenge:

- The code translation needs to take the paradigm shift into consideration
  - Java is object-oriented language model
  - FaaS is inherently bound to the functional programming paradigm which is stateless computation with strict use of invocation parameters and return values without global variables

Programming → Execution Model Mapping

Second challenge:

- Mapping of Java class to appropriately packaged Java FaaS functions
  - Empty methods, getters and setters, constructors and singletons
  - Typical Java project conventions such as src folder and the absence thereof and exceptions from the convention

Programming → Execution Model Mapping

Third challenge:

- The mapping needs to consider the grouping of method per functional unit
  - To avoid excessive network calls
  - To ensure that all dependency methods referenced from each method can be resolved

State Handling

Two ways to handle the state of resulting decomposed functions

- Extending the method signature to pass in and out all attributed
- Using server-side state

Podilizer uses the first approach after weighting the advantages of extended method signature (price, functional purity) against S3 (performance)

Critique: No details about how to measure these features

State Handling

- Stateful Java methods → Stateless function unit
- Making self-references explicit by enhancing the method signatures with it
  - In Java, we use ***Class.method(params)*** to change the state of instances, the instance is self-referenced implicitly with the keyword ***this***

  The translation process rewrites the method with Lambda-required signature and generated code → initialises the invocation credentials and creates an input object to save instance state → initialises the Lambda invoker with the input object → calls the ***Class.handleRequest(input, output, context)*** → fetches the result from the output object → renews the instance state using the result object

FaaSification Pipeline

- A: static code parsing and **analysis**
- D: **decomposition** into functional units
- F: source-to-source **translation of the functional units** into FaaS units, adhering tyo the calling conventions of the target platform
- C: **compilation** and dependency assembling of these units
- U: **upload**, deployment and configuration of these units
- V: systematic test and **verification**

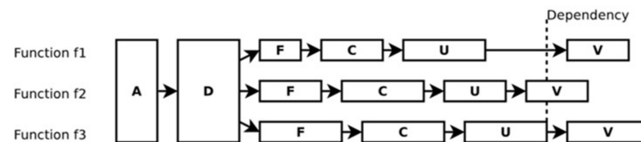


Figure 1: FaaSification pipeline

Podilizer Design

- The pipeline thinking is reflected in the design of Podilizer
  - Podilizer recursively scans directories for Java projects and processes each project and source file until the code is available to be invoked as Lambda function
  - The pipeline steps are incremental and the tool allows for continuations starting from each step
  - This design makes it fault-tolerant and debug-friendly

Table 1: FaaSification pipeline steps and continuations.

Step	Continuation Points	Requirements
A: code analysis	– (internal AST)	source code directory
D: code decomposition	– (internal list of ASTs)	–
F: function translation	target source directory	–
C: compilation	target binary directory	Maven buildfile (cust.)
U: upload	deployed functions list	AWS credentials
V: verification	–	unit test definitions

Podilizer Implementation

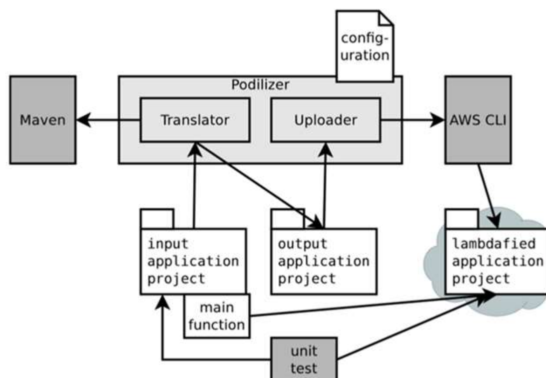


Figure 2: Implementation architecture of Podilizer

Trials and Findings

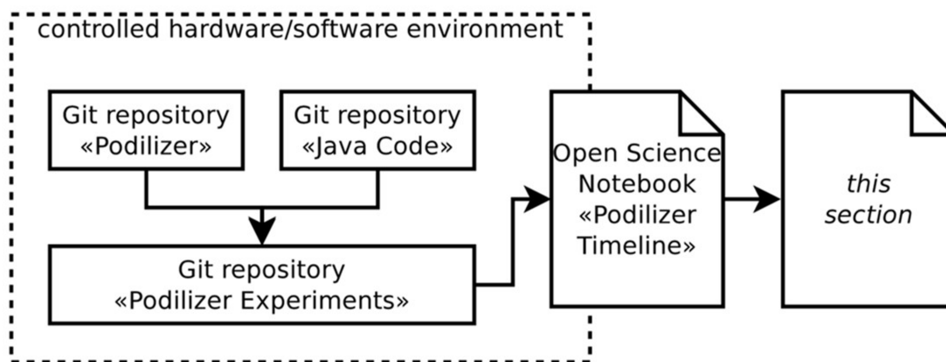


Figure: Testbed for performing experiments on Podilizer

Experiment Setup

Each step has their unique check

The first three steps are performed internally by Podilizer.

The three remaining ones are merely automated by running executables out of which one is provided by Podilizer

Step	Check
A: code analysis	JavaParser internal return value
D: code decomposition	Podilizer internal
F: function translation	Podilizer internal
C: compilation	compiler/build tool exit status
U: upload	Podilizer deployer exit status
V: verification	call test, unit test exit status

Performance Experiment

Podilizer is instrumented with millisecond-precision logging to reveal the duration of each pipeline step. The performance, the quality of the transformation, can be measured by the ratio of successful checks against all which are performed in each step.

The economic aspect comparison should be calculated manually, for the absent of a general performance estimation formula.

Input property

In the paper, we set 6 types of applications.

1: Graphical window with buttons, User interface
2: Mathematical function
3: Calculation of shipping containers and boxes
4: Public transport information
5: Image processing
6: Specific language parsing and evaluation

Result

The result can be impacted by hardware, used software tools, the provide, and network connection.

Hardware support:

We run the experiment on Dell Latitude E7450 notebook, Intel Core i7-5600 quad-core processor clocked at 2.60GHz, connected to SWITCHlan, the Swiss university network, via 1000baseT Ethernet. The computer installed with Ubuntu Linux and OpenJDK 8.

The performance and quality of The FaaSification pipeline for P1

Step	Performance	Quality
A: code analysis	0.055s	100%
D: code decomposition	0.002s	100%
F: function translation	0.122s	100%
C: compilation	10.173s	100%
U: upload	21.238s	100%
V: verification	–	–
TOTAL	31.590s	success

Performance of P2 to P6

Step	P2:P	P3:P	P4:P	P5:P	P6:P
A	0.054s	0.058s	0.074s	0.074s	0.105s
D	0.002s	0.005s	0.010s	0.011s	0.028s
F	0.096s	0.302s	0.867s	0.025s	0.701s
C	10.530s	17.777s	37.707s	–	22.901s
U	21.349s	31.141s	65.075s	–	44.858s
V	11.942s	–	13.927s	–	–
TOTAL	43.973s	49.283s	117.657s	–	68.593s

FaaSification pipeline quality for P2-P6

Step	P2:Q	P3:Q	P4:Q	P5:Q	P6:Q
A	100%	100%	100%	100%	100%
D	100%	100%	100%	100%	100%
F	100%	100%	100%	0%	100%
C	100%	100%	100%	0%	100%
U	100%	100%	100%	0%	100%
V	100%	–	100%	–	–
TOTAL	success	success	success	fail	success

Performance analysis

The first 2 steps almost execute in less than a single Lambda billing period.

The compilation and upload always take much more longer time in comparison.

P5 failed due to a crash of the crash of the transformator itself.

The automated translation is feasible, with high code coverage for simple and heterogeneous code projects. The failure are due to the dynamic classloading for plugins and the insufficient handling of such constructs by the transformer.

Execution performance of applications

Flavour	P1:X	P2:X	P3:X	P4:X	P5:X	P6:X
Notebook local	–	0.71s	1.87s	1.25s	0.08s	0.13s
AWS EC2 local	–	1.18s	2.99s	1.92s	0.09s	0.18s
AWS EC2 Xinetd	–	1.16s	2.86s	1.57s	0.12s	0.22s
AWS Beanstalk	–	0.36s	0.36s	1.79s	–	0.36s
AWS Lambda	–	8.77s	9.74s	12.20s	–	–

Results

P1 fails due to being a graphical application. P5 requires an interactive command-line interface and cannot run in a web environment or through function calls alone. P6 fails for missing symbol files for the parser.

EC2 instance is always slower than Notebook local. Two layer of indirection through Xinetd and a wrapper shell scripts do not often cause higher execution time.

The results gives an answer to RQ1: While the applications perform slower by about an order of magnitude compared to typical IaaS or PaaS deployments, the economic feasibility is still in range for services which are not permanently invoked.

Comparison between Source code size before and after

Flavour	P1:S	P2:S	P3:S	P4:S	P5:S	P6:S
Original	20 kb	32 kb	44 kb	40 kb	40 kb	96 kb
Lambdafied	548 kb	12436 kb	988 kb	2960 kb	–	1796 kb
Overhead	2640%	38763%	2145%	7300%	–	1771%

Conclusion

- Promising for future cloud application engineering
- Beneficial to programming education for simple OOP
- Code which is not prepared for individual function access will originate problem
- Interface with JVM will also be a future difficulty
  - Through the classloader or the CLI, as well as data access from different file paths

Strength

- Innovation to decompose legacy application to FaaS
- Acceptable success rate for the existing experiment
- Multi-environment
- Multi kind of application

Weakness

About the experiment

- Lack of evidence to prove extended method signature is better than S3.
- What kind of resource consumption superpositioning and dependency can support parallel execution

Weakness

About the result

- Runtime
- Code Size
- Application limitation
- Can only deal with the most simple application
- Only support lambda and Java

Evaluation

- Acceptable result and success rate
- Indeed create a tool to auto-deploy application to lambda
- Lack of evidence
- Need more experiment to be more convincing

Future Work

- More Serverless Computing Platform
- More Language support
- Performance can be better
- Optimize attribute and method dependency
- Support more type of application