View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ZHAW digitalcollection

Selective Java code transformation into AWS Lambda functions

Serhii Dorodko and Josef Spillner
Zurich University of Applied Sciences,
School of Engineering, Service Prototyping Lab (`blog.zhaw.ch/splab`), Switzerland
`{dord,josef.spillner}@zhaw.ch`

Abstract

Cloud platforms offer diverse evolving programming and deployment models which require not only application code adaptation, but also retraining and changing developer mindsets. Such change is costly and is better served by automated tools. Subject of the study are automated *FaaSification* processes which transform conventional annotated Java methods into executable Function-as-a-Service units. Given the novelty of the problem domain, a key concern is the demonstration of feasibility within arbitrary boundaries of FaaS offerings and the measurement of resulting technical and pricing metrics. We contribute a suitable tool design called Termite with corresponding implementation in Java. The design is aligned with a generic transformation pipeline in which each step from code analysis over compilation to deployment and testing can be observed and measured separately. Our results show that annotations are suitable means for fine-grained configuration despite ceding control to the build system. Smaller Java projects can be FaaS-enabled with little effort. We expect FaaSification tools to become part of build chains on a wide scale once their current engineering shortcomings in terms of tackling more complex code are solved.

1 Motivation

New computing paradigms such as cloud computing bring along new programming models with high frequency. In domains centered around discrete events such as connected devices, cloud automation or electronic markets, trends indicate that the execution of short-lived functions in Function-as-a-Service (FaaS) environments will become the dominant hosted code paradigm. As the FaaS programming, deployment and execution model is becoming more popular, its exploitation would benefit from a developer-controlled semi-automated transformation of legacy code and of generic new code to the coding and packaging conventions expected by this model, including runtime restrictions. Ideally, programmers would not have to worry about how to map Java classes into cloud functions; rather, a smart transformation mechanism would perform this work, resembling a compiler which would treat the function runtime as its low-level target platform.

In our previous work, we have reported on a fully-automated approach called Podilizer [SD17] which however fails to transform a number of functions or methods. Some failures occur at translation time due to complex code structure. Possible reasons for runtime failures beyond translation are use of local input from files or interactive terminals. Furthermore, the runtime overhead of Podilizer becomes needlessly high whereas in practice, only few functions on the performance-critical path may need to be offloaded to a FaaS environment.

Instead, in this work, we explore an assisted approach which is controlled by programming language annotations and introduce Termite as tooling support. As with the previous work on Podilizer, we limit the study and observations to Java code as input and AWS Lambda functions as output in order to allow for comparison, but claim that most of the observations also apply to other combinations without loss of generality.

Hence, we aim at contributing novel insight into a selective automated transformation of legacy applications into FaaS-hosted cloud applications which we name *FaaSification* for the general case from a software technology perspective. We claim that our tools, Podilizer and Termite[1], are the first ones which perform such a transformation from monolithic Java code to AWS Lambda units, and do so with sufficient qual-

[1]Podilizer is publicly available at `https://github.com/serviceprototypinglab/podilizer` and Termite at `https://github.com/serviceprototypinglab/termite`.

CEUR-WS.org/Vol-2330/paper2.pdf

ity to be considered in cloud application prototyping projects. Due to the restriction to Lambda, we refer to this process more specifically as *Lambdafication*. We employ an experimental scientific method with synthetically generated and manually engineered applications. The research is backed up by a curated dataset containing publicly verifiable tools, reference applications and experiment instructions. As we have previously reported extensively on Podilizer [SD17], we will not repeat its design and only include the tool for the purpose of experimental comparison in this article.

The article is structured as follows. First, it discusses related work. Then, it outlines the research questions and the chosen approach and explain how to map object-oriented programming to a function-based service and execution model. The mapping description is complemented by an abstract pipeline architecture and a concrete realisation thereof, the Termite tool, with unique benefits in the trade-off between automation and developer control. Afterwards, we share the experimental evaluation results and discuss the extracted findings. The paper concludes with an open discussion of how software should be written for the cloud based on fine-grained functions.

2 Related Work

Software engineering for FaaS environments is an emerging topic. In 2019, the European research project RADON was initiated to produce an advanced DevOps framework for FaaS but will require time to produce first results. As one of the first large-scale academic efforts, it contrasts established industry approaches such as the Serverless Framework. In recent versions of this framework, cloud function code is supplied with wrapper functions to achieve deployment across cloud providers. No further code analysis is performed, and the framework assumes a function-native development approach.

Code transformation has been proposed to reduce software prototyping and engineering effort for several new execution environments. In 2015, Cai et al. introduced a pattern-based technique for cloud-readiness [CZW+15]. The technique uses patterns, rules and templates and was validated on 19 open source Java projects migrated to AWS cloud services. The transformation did not change the execution nature but modified the storage/persistence layer to attach to DynamoDB and other cloud database and security services. No performance numbers or other details beyond the changed lines of code were reported.

Already in 2012, JTransformer was proposed to achieve higher precision and recall of design pattern detection in code analysis and code reengineering tasks [BK12]. It is available as plugin for Eclipse JDT. It works on factbases which are associated to Prolog processes to produce analyses and transformations (A&T). While a useful expert tool, it requires knowledge of Prolog to write A&T which is not a skill most FaaS developers would possess, and it lacks a runtime integration for applications with selective offloading to cloud functions.

3 Approach

Our approach consists of three parts. First, we identify three research questions. Then, we explain general decisions which must be taken by any transformation tool related to the programming model, the handling of stateful objects and the design of a transformation process. The third part presents both the design and the implementation of our transformation tool Termite.

3.1 Research Questions

The planned code transformation process leads to three research questions (RQ_1–RQ_3). Our approach is focused on generating empirical results and deriving the answers accordingly. Both RQ_1 and RQ_2 mirror the research questions from our initial work on Podilizer but will result in different answers due to our advanced approach.

RQ_1: Is it economically viable to run a Java application entirely over FaaS with no or little cost overhead for typical usage patterns? The comparison baseline would be conventional programmable platform (PaaS) models by deploying the application onto a suitable application server as well as programmable infrastructure (IaaS) by wrapping the Java application into a container or a virtual machine. Evidently, the question needs to be broken down to the granularity of offloading Java methods and the way they interact with each other. If each small helper function becomes a cloud function under the FaaS pricing model, deeper callgraphs will quickly result in prohibitively expensive hosting.

RQ_2: Is it technically feasible to automate part of this process? And if so, which percentage of code coverage can be expected, which performance can be

achieved, and which code is easier, hard or impossible to convert? Evidently, due to our approach of using annotations, the automation depends on the choice of where to put which kind of annotation.

RQ_3: Is there a friction-free integration with established Java development notations and processes? Can the FaaSification be integrated into existing build tools?

3.2 Programming $\mapsto$ Execution Model Mapping

FaaS is inherently bound to the functional programming paradigm. Its characteristics under a pure interpretation are determined by stateless computations with strict use of invocation parameters and return values without global variables. In practice, just as functional programming languages have introduced techniques to cause side effects and manage state, for instance through monads, so do most FaaS interfaces, for instance through access to storage services.

The function orientation is in contrast to Java's model which is predominantly an object-oriented language. Even though few functional programming concepts are available through the Functional Java library and starting with Java 8 with native Lambda expressions [Plu14], the dominant share of code is written in a pure object-oriented way. The same observation applies to similar programming language. Therefore, the code translation needs to take the paradigm shift into account.

The challenges are then related to the mapping of Java classes to appropriately packaged Java FaaS functions, called FaaS units or functional units. The mapping needs to account for empty methods, getters and setters, constructors and singletons. Beyond the code, typical Java project conventions such as the presence of a `src` folder, but also the absence thereof and exceptions from the conventions, need to be accounted for. Finally, the mapping needs to consider the grouping of methods per functional unit to avoid excessive network calls and ensure that all dependency methods referenced from each method can be resolved.

Considering execution cost (RQ_1), in commercial FaaS environments the three contributing factors are execution time, memory allocation and a fixed per-invocation fee, with some providers speeding up execution upon allocation of more memory, while code size is not a factor. Applying code transformation could lead to combined cost/performance improvements by packing various combinations of local functions into cloud function bundles and invoking the appropriate bundle. In our approach, we perform a simple 1:1 mapping instead.

Considering the automation (RQ_2), we note that using annotations allows for configuring per-method parameters on where and how the cloud function should be executed, including an individual assignment of memory allocation and runtime constraints. Moreover, we argue for a fine-grained staged code transformation so that analytical and corrective measures can be introduced for each method in various stages. In our approach, we will use a pipeline model with multiple transformation steps per function, and an injection of additional code for transparently handling state.

3.3 State Handling

Java methods are often stateful through instance attributes. The state handling of the resulting decomposed functions can either extend the method signature to pass in and out all attributes that are accessed and modified, or use server-side state. AWS Lambda offers both an S3 blob storage interface and local environment variables. However, the variables are restricted to read-only access from the runtime. Weighting the advantages of S3 (performance) against extended method signatures (price, functional purity), our approach uses the latter technique.

In Java, methods with parameters are integral parts of classes and are used to change the state of the corresponding instances: $Class.method(params)$. The instance is self-referenced implicitly with the keyword *this*. According to the Lambda programming model, every function unit assumes a specific stateless class with a method handler which is triggered when the function is invoked. The statelessness is due to not guaranteeing the same object of this class to be used for subsequent requests. Early works to compile Java code into a typed Lambda calculus have suggested making the self-references explicit by enhancing the method signatures with it [WJUH98] which is the approach chosen by us as well.

The translation process thus rewrites the method header with the Lambda-required signature and the method body with generated code. This code first initialises the invocation credentials, creates an input object to save the instance state as well as any method parameters, initialises the Lambda invoker with the created input object se-

rialised to JSON, calls the method (*Class.handleRequest*(*input*, *output*, *context*)), fetches the result from the deserialised output object, and renews the instance state using the result object. The credentials are read from the environment and upon failure from a configuration file which permits the generated code to still run outside of the Lambda environment.

3.4 FaaSification Pipeline

FaaSification is the process of converting a code structure into a format that is executable on FaaS. In our approach, inspired by the predecessor work [SD17], this process is represented by a superscalar pipeline which allows for parallel processing of each of its steps. The first step is the static code parsing and analysis (A). The second step is the decomposition into functional units and a remainder which includes the code identified as incompatible to the target FaaS platform (D). The third step is the source-to-source translation of the functional units into FaaS units, adhering to the calling conventions of the target platform (F). The fourth step is the compilation and dependency assembling of these units (C), and the fifth step is their upload, deployment and configuration to turn them into ready to use microservices (U). An optional sixth step is the systematic test of all deployed functions and the verification of the successful transformation (U). Fig. 1 gives an example of a generic FaaSification pipeline whose parallel execution depends on resource consumption superpositioning and on dependencies between methods before the ability to run unit tests. In our work, we refer to the process adopted for targeting AWS Lambda within the boundaries of the pipeline more concretely as Lambdafication.

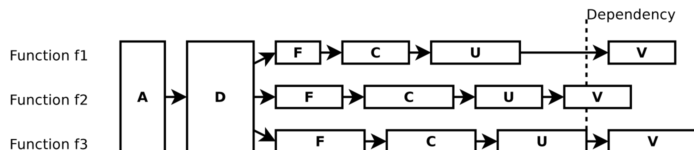

Figure 1: FaaSification pipeline

4 Tool Design and Implementation

Following a research approach based on competing designs, we design two distinct tools with different tradeoffs between developer effort and technical characteristics of the resulting functions. The first, Podilizer, attempts to lambdafy an entire Java project unconditionally, including all classes and public methods. Any configuration is read from a per-project configuration file. Podilizer has been described in detail in previous work [SD17]. The second, Termite, only lambdafies methods explicitly marked by the software developer. Any configuration can be customised through the markers. The impact of the competing designs on software development contributes to the answer for RQ_3. We are interested to quantify the differences between the two tools via conformance and performance measurements to achieve an answer for RQ_2. The resulting functions are compared against their original monolithic implementations to finally answer RQ_1.

4.1 Termite Design

In contrast to Podilizer, Termite supports the selective publishing of functions through Java method annotations. Through parameters to these annotations, the deployment as well as operational aspects can be controlled by the software engineer which aligns with combined developer and operator roles (DevOps). Using annotations offers the following advantages: They are part of the Java language syntax, their vocabulary can be learned easily as domain-specific language, and they can be extended to cover future FaaS features. In Termite, @Lambda annotations are thus applied on the method level, although recent research on @Java for more fine-grained annotations suggests that more sophisticated FaaS deployments might become possible [CV13]. All parameters have default values and are therefore optional, contributing to a rapid service prototyping experience. The set of supported parameters is described in Table 1.

Table 1: Lambda annotation parameters.

Parameters	Semantics	Default
endpoint	endpoint URL, selection of runtime provider	`null`
region	deployment region	us-west-2
memory	execution memory limit in MB	1024
timeout	execution timeout in seconds	60
env	environment variables available to instance	–

A unique feature is the ability to define the endpoint. It allows for deployment and invocation of functions at alternative FaaS installations that offer the AWS Lambda interface. The choice of provider is thus made programmable as well, reducing vendor lock-in.

Fig. 2 represents the compilation pipeline when using Termite in software development projects. Compared to the generic pipeline, several steps are either combined or subdivided, ceding control over the order

and parallelisation of method processing. Termite's transformation work is hooked into the compilation before the Java compiler performs its work. Its runtime part to invoke dependency functions is furthermore coded into the rewritten method invocations ($I2$).

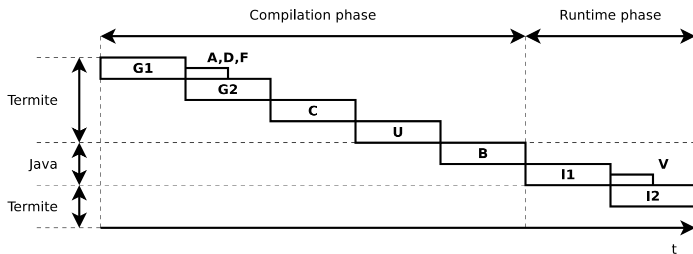

Figure 2: Termite compilation pipeline: Generation of functions (G1) and additional sources (G2), compilation (C) and upload (U) of functions, build (B) and invocation (I1) of the annotated Java application, and hosted function invocation (I2)

4.2 Termite Implementation

Besides the Lambda function source code generation, Termite takes care of invocations of each function at runtime. The implementation uses AspectJ and the Reflection API. The purpose of the AspectJ library is to create aspects, basically interceptors. Custom aspects can be configured to react on a particular annotation, to interrupt the normal runtime workflow through join points and to perform custom actions instead. In Termite, the interception logic mandates that each method annotated with the @Lambda annotation is being intercepted and then separated into a new thread. Lambda's concurrency paradigm is thus exploited. Going deeper into the interception logic, aspects can interrupt the workflow before, after and around every method or particular marked ones, for instance with certain annotation. Termite uses the "around" interception, meaning that aspects will be called instead of the main method which is then in turn invoked again from the aspect. The aspect initialisation is demonstrated in Listing 1.

The aspect's join point allows to use the Java Reflection API to get the method specification and perform the invocation. Furthermore, if a hosted function is unreachable, a configuration option allows for the method to be called locally, providing a fail-over mechanism for functions that are outsourced for performance reasons without consideration for local code size.

Listing 1: Use of aspects within Termite.

```
@Aspect
public class Invoke {
  @Around("@annotation(lambda)")
  public Object anyExec(ProceedingJoinPoint joinPoint,
  ch.zhaw.splab.termite.annotations.Lambda lambda)
  throws Throwable {
    MethodSignature signature = (MethodSignature)
    joinPoint.getSignature();
    Method method = signature.getMethod();
    InvokeThread invokeThread = new InvokeThread(method,
    lambda, joinPoint);
    invokeThread.start();
    return null;
  }
}
```

The resulting implementation architecture and workflow of Termite is shown in Fig. 3.

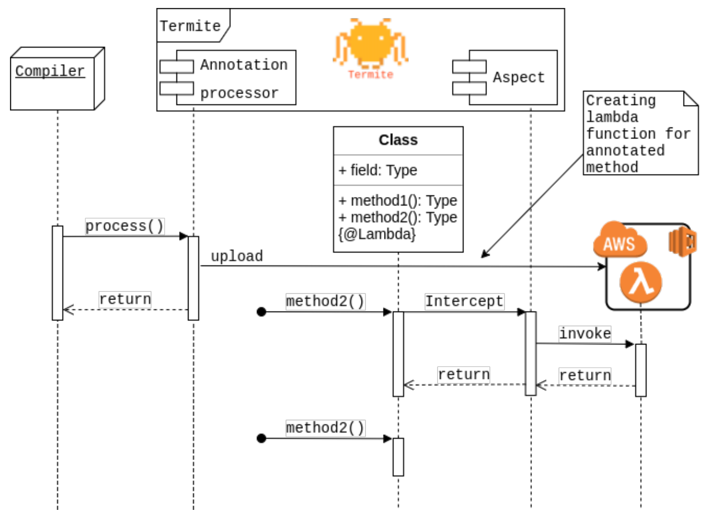

Figure 3: Termite implementation architecture and workflow

Projects compiled with Termite are modified to depend on Termite's invocation module. Any data exchange between local and functions is performed using JSON through this module. For each generated Lambda function, additional input and output classes are also generated. These classes are usual POJOs, resulting in plain old java objects without extending existing classes or using other annotations. The input object is mapped into JSON and transports information to the Lambda function instance. The object of type output brings data back to the invoker. Input and output classes are built based on the method's specification, its parameters for the input type and its return value for the output type. Additionally, the output type contains fields for service time and environment description. Listing 2 gives an example of how a summation method is transformed by Termite.

Listing 2: Example of a Termite code transformation.

```
@Lambda()
public static int sum(int a1, int a2){
        System.out.println("a1 + a2 = " + (a1 + a2));
        return a1 + a2;
}

// is transformed into (with simplified imports)

import java.io.*;
import com.amazonaws.services.lambda.runtime.*;
import com.amazonaws.util.IOUtils;
import com.fasterxml.jackson.databind.*;

public class LambdaFunction implements RequestStreamHandler{
  public void handleRequest(InputStream inputStream,
  OutputStream outputStream, Context context) throws
  IOException {
    long time = System.currentTimeMillis();
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.disable(SerializationFeature.FAIL_ON_
    EMPTY_BEANS);
    String inputString = IOUtils.toString(inputStream);
    InputType inputType = objectMapper.readValue(inputString,
    InputType.class);
    int result = sum(inputType.getA1(), inputType.getA2());
    OutputType outputType = new OutputType("Lambda
    environment", System.currentTimeMillis() - time, result);
    objectMapper.writeValue(outputStream, outputType);
  }

  public static int sum(...) {...} // as above
}
```

4.3 Termite Limitations.

The implemented prototype only supports synchronous Lambda function calls which may change the program semantics due to Java offering an asynchronous FutureTask class for concurrent compute tasks. Asynchronous methods using FutureTask would need to enforce the corresponding asynchronous function calls to avoid blocking, even though they do not offer cancellation, making a strict translation impossible. The mismatch is caused by Lambda not treating function instances as first-class citizens. Once deployed, changes in methods are not recognised and do not automatically lead to a re-upload. This means that even for smaller changes, a forced re-deployment has to be performed to avoid code inconsistencies.

5 Trials and Findings

We evaluate Termite experimentally with a standard research testbed approach shown in Fig. 4, first on its own and then in comparison to Podilizer. All input to the experiments is recorded in public versioned repositories, and all raw output is captured in another dedicated repository. The versioning allows for finding improvements and regressions over time as the software evolves. All experiments are tracked in a public Open Science Notebook and tools for reproducibility, repeatability and recomputation are made available as well in a Lambdafication Repeatability project[2]. The main results of the current implementation are reflected in this section.

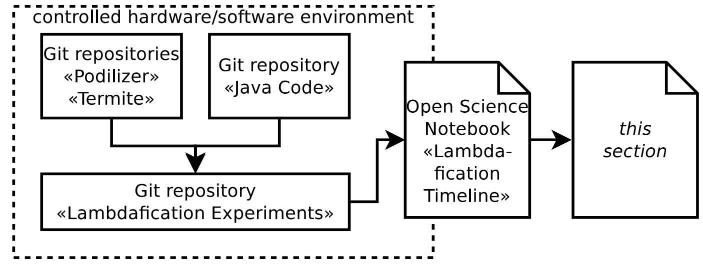

Figure 4: Testbed for performing experiments on Podilizer and Termite

5.1 Experiment Setup

Each step of the pipeline is associated to a unique check for success. The first three steps are performed internally by Podilizer within one procedure, whereas the three remaining ones are merely automated by running executables out of which one is provided by Podilizer, too. In Termite, only the third step is performed by the tool and the last two steps are automated. The most crucial check is the final one which is successful if all deployed functions are remotely invocable. According to DZone, about 30.7% of all Java projects hosted on Github depend on JUnit which calls for an integration to ensure systematic testing of the deployment [Har13]. Table 2 summarises all steps and checks.

Table 2: Lambdafication pipeline steps and checks.

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

Both transformation tools instrumented with millisecond-precision logging to reveal the duration of each pipeline step. In addition to 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 requires a comparison between the execution of the lambdafied application compared to a monolithic execution in a configuration which matches the performance. In the absence of a general

[2]Lambdafication Repeatability in the Open Science Framework: https://osf.io/c886p/

performance estimation formula, a manual calibration specific to each software application under test is therefore needed.

The reference input project set consists of six software applications which represent the large variety of Java software engineering, ranging from 28 to 771 significant lines of code (SLOC), diverse interaction forms (none, standard input and output, graphical, files, HTTP methods) and build tools (javac, make, maven, ant) and artefact type (applications, libraries, plugins, tests). The software projects are a graphical window with buttons (P1), mathematical functions (P2), calculation of shipping containers and boxes (P3), public transport information (P4), image processing (P5) and domain-specific language parsing and evaluation (P6). An artificial project consisting of 100 numbered Java hello world methods is used as additional comparison point (P7).

5.2 Results

We have run the experiment on a notebook with Intel Core i7-4800MQ quad-core processor clocked at 2.70 GHz. The notebook was connected to SWITCHlan, the Swiss university network, via 1000baseT Ethernet, and installed with Ubuntu Linux and OpenJDK 8. The results differ depending on the chosen software project to translate. The values are also influenced by the hardware, the used software tools (Termite and Podilizer, AspectJ, Maven, JUnit), the FaaS runtime environment (AWS Lambda, Snafu) and the network connection in between. A full specification and self-contained virtual machine is made available as part of the Lambdafication Repeatability project.

5.2.1 Termite results

The evaluation of Termite follows the same pattern as the one already published for Podilizer. Table 3 summarises the achieved transformation quality on the projects P1–P6. Due to the combined analysis and transformation steps, the results for $A \rightarrow D \rightarrow F$ are joined. The evaluation omits the verification step V entirely due to the impossibility to invoke remote functions from other functions with the current implementation. Consequently, all projects fail overall although porting the related functionality from Podilizer would be possible with moderate engineering effort beyond our research work.

The original project sizes after adjustments to compile with Termite through Maven and the resulting sizes with generated code are compared in Table 4. Compared to Podilizer, the overhead is much lower both on average and per each individual project.

Table 3: Lambdafication pipeline characteristics (quality) for P1–P6 with Termite.

Step	P1:Q	P2:Q	P3:Q	P4:Q	P5:Q	P6:Q
A,D,F	100%	100%	100%	100%	100%	100%
C	0%	33%	8%	0%	6%	0%
U	0%	33%	8%	0%	6%	0%
V	–	–	–	–	–	–
TOTAL	fail	fail	fail	fail	fail	fail

Table 4: Application source code size comparison before/after using Termite.

Flavour	P1:S	P2:S	P3:S	P4:S	P5:S	P6:S
Original	32 kb	44 kb	44 kb	44 kb	48 kb	100 kb
Lambdafied	332 kb	492 kb	1528 kb	1288 kb	1964 kb	736 kb
Overhead	938%	1018%	3373%	2827%	3992%	636%

5.2.2 Comparative results

Fig. 5 shows the performance timeline of Termite on the synthetic application P7. On average, including the Maven overhead, around 5-6 seconds are spent on processing each annotated function in addition to possible network delays. The total processing time per method is 12.06 s which factors in further Maven and AspectJ overheads. In contrast, Fig. 6 shows the corresponding performance timeline of Podilizer. The raw build and upload time is similar, albeit slightly higher compared to Termite with about 6-7 seconds. The discrepancy can be observed with the translation time that the tool implements by itself which slows it down considerably to a total processing time per method of 17.21 s. This implies that despite Termite's more flexible approach, it is about one third faster than Podilizer. The drawback is that the Termite implementation does not consider dependency functions whereas Podilizer considers all methods in a file as potential dependencies for which proxy stubs are generated.

Table 5 compares the overhead on code size of both approaches, starting with the synthetic code file of P7 which accounts for 14.0 kb with annotations and 5.8 kb without. Due to the included stubs, the overhead is much higher on the source side with Podilizer, but less on the binary side as the dependency JARs per function, which are almost equal in both tools and include the AWS SDK, account for the majority of added space. An apparent drawback is that Podilizer assumes all other methods per file to be dependencies instead of performing a static call tracing which would drastically

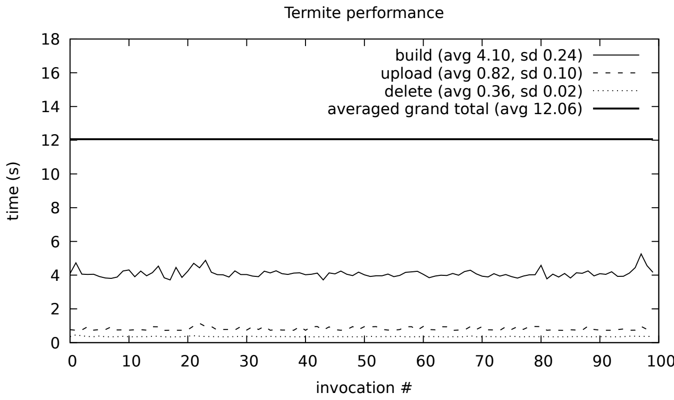

Figure 5: Performance timeline of Termite on P7

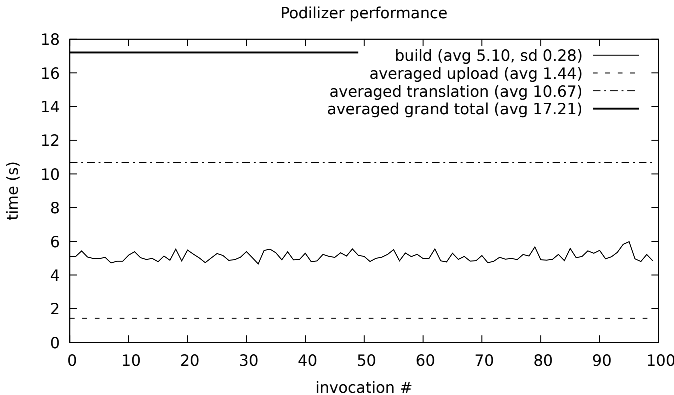

Figure 6: Performance timeline of Podilizer on P7

reduce the overhead and could then be implemented in Termite as well, reducing the differences between both approaches on the implementation level.

Table 5: Application source and binary code size.

Flavour	P7:S	P7:B
Original	5.8/14.0 kb	6.2 kb
Lambdafied with Podilizer	89707 kb	604809 kb
Lambdafied with Termite	8391 kb	534573 kb
Podilizer overhead factor	15465	97549
Termite overhead factor	598	86220

6 Discussion

Our findings in automated Java code to Lambda units transformation look promising for future cloud application engineering. The results are also beneficial to programming education where rather simple object-oriented applications are in wide use and educators regularly struggle to keep up with new application hosting formats and platform services.

Difficulties originate from code that is not prepared for individual function access. According to a recent study, at least 20% of Java methods are too accessible (*public* instead of *protected* or *private*) [VBMP16], while for our work, they are sometimes too inaccessible, although the solution to both is the same: powerful refactoring tools for software engineers. Further difficulties originate from interfacing with the Java virtual machine, for instance through the classloader, and with the command-line interface, as well as with data access with differing file paths.

A comparison between our two approaches, Podilizer and Termite, is available through Table 6. The commonalities in scope and process are contrasting the differences in how much control a developer has over the process. The advantages of the Termite which included lessons learned from the earlier Podilizer implementation are clear. Its design foresees for instance the fallback invocation of the local method in cases where the cloud-deployed transformed copy is not available.

Table 6: Comparison of approaches and tool designs.

Approach	Language	Selective	Configurable	Failsafe
Podilizer	Java	no	only through AWS CLI	no
Termite	Java	yes	yes (annotations)	yes

The resulting implementations of the same name are further compared in Table 7. While Termite is limited to rather simple code structures, Podilizer handles several advanced cases well including exceptions and dependencies between methods, but also weighs in with more than twice the significant Lines of Code (LoC). Both tools integrate with Maven and handle common programming styles such as classes with and without packages, but Termite currently lacks proper support for object attributes.

Table 7: Comparison of implementation.

Implementation	Deps	Exceptions	Attributes	LoC
Podilizer	all	transformed	transferred	2500
Termite	none	not considered	not considered	1030

With the collected data, answering the research questions becomes possible. While the answers have been discussed in context before, the following list summarises the claims.

Answer to RQ_1: The economic viability of FaaS depends on the request frequency and load in addition to the rather volatile pricing model of cloud providers. In the case of the reference applications, the load generally justifies the FaaSification when less than 200000 requests per month occur.

Answer to RQ_2: The automation is possible al-

though keeping the engineer in the loop through controlling annotations and a resulting semi-automation is preferable. The averaged code coverage ranges from 8% (Termite) to 83% (Podilizer) due to implementation differences. The averaged overhead figure ranges from 2131% (Termite) to around 10518% (Podilizer), and the performance figure from 62 s (Podilizer) to 254 s (Termite).

Answer to RQ_3: Annotations are a suitable established Java notation which given the current progress of the field represent the least friction in terms of development tooling integration. Triggering the FaaSification process through the build system is also considered suitable, even though alternatives such as explicit method calls at runtime have not been discussed.

Overall, we believe that the tool designs and implementations are helpful in accelerating cloud deployments despite needing more fundamental research on the OOP to FaaS mapping and tool engineering and testing. Inspired by the results of using Termite, we have successfully implemented a basic form of selective FaaSification to the Lambada tool which performs static and dynamic transformation for Python applications [Spi17].

Future work identified by limitations of our approaches encompasses the handling of dynamic classloading, server-side state handling, FaaSification beyond Java and Python as input languages and AWS Lambda as target service, improved developer tooling integration, as well as optimisations for attribute and method dependencies including the creation of function clusters or bundles.

Acknowledgements

This research has been supported by an AWS in Education Research Grant which helped us to run our experiments on AWS Lambda as representative public commercial FaaS.

References

[BK12] Alexander Binun and Günter Kniesel. Joining Forces for Higher Precision and Recall of Design Pattern Detection. Technical Report IAI-TR-2012-01, University of Bonn, January 2012.

[CV13] Walter Cazzola and Edoardo Vacchi. @Java: annotations in freedom. In *28th Annual ACM Symposium on Applied Computing*, pages 1688–1693, Coimbra, Portugal, March 2013.

[CZW+15] Zhengong Cai, Liping Zhao, Xinyu Wang, Xiaohu Yang, Juntao Qin, and Keting Yin. A pattern-based code transformation approach for cloud application migration. In *8th IEEE International Conference on Cloud Computing, CLOUD 2015, New York City, NY, USA, June 27 - July 2, 2015*, pages 33–40, 2015.

[Har13] Chen Harel. GitHub's 10,000 Most Popular Java Projects: Here are the Top Libraries They Use. online: `https://dzone.com/articles/githubs-10000-most-popular`, 2013.

[Plu14] Martin Pluemicke. Functional Interfaces vs. Function Types in Java with Lambdas. In *Software Engineering (Workshops)*, number 1129 in CEUR-WS, pages 146–147, Kiel, Germany, February 2014.

[SD17] Josef Spillner and Serhii Dorodko. Java Code Analysis and Transformation into AWS Lambda Functions. arχiv:1702.05510, February 2017.

[Spi17] Josef Spillner. Transformation of Python Applications into Function-as-a-Service Deployments. arχiv:1705.08169, May 2017.

[ST15] Nikita Salnikov-Tarnovski. Java version statistics: 2015 edition. online: `https://plumbr.eu/blog/java/java-version-statistics-2015-edition`, 2015.

[VBMP16] Santiago A. Vidal, Alexandre Bergel, Claudia Marcos, and J. Andrés Díaz Pace. Understanding and addressing exhibitionism in Java empirical research about method accessibility. *Empirical Software Engineering*, 21(2):483–516, April 2016.

[WJUH98] Andrew K. Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a Typed Lambda-Calculus: A Preliminary Report. In *Types in Compilation – 2nd International Workshop*, volume 1473 of *LNCS*, pages 9–27, Kyoto, Japan, March 1998.