Beyond @CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns

Raffael Klingler
Zurich University of Applied Sciences
Winterthur, Switzerland

Nemanja Trifunovic
Zurich University of Applied Sciences
Winterthur, Switzerland

Josef Spillner
Zurich University of Applied Sciences
Winterthur, Switzerland

ABSTRACT

Simplicity in elastically scalable application development is a key concern addressed by the serverless computing paradigm, in particular the code-level Function-as-a-Service (FaaS). Various FaaSification frameworks demonstrated that marking code methods to streamline their offloading as cloud functions offers a simple bridge to software engineering habits. As application complexity increases, more complex runtime patterns with background activities, such as keeping containerised cloud functions warm to ensure the absence of cold starts, usually require giving up on simplicity and instead investing efforts into orchestrating infrastructure. By bringing infrastructure-as-code concepts into the function source via powerful code annotations, typical orchestration patterns can be simplified again. We evaluate this idea and demonstrate its practical feasibility with FaaS Fusion, an annotations library and transpiler framework for JavaScript.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

serverless computing, cloudware, software engineering, deployment

ACM Reference Format:
Raffael Klingler, Nemanja Trifunovic, and Josef Spillner. 2021. Beyond @CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns. In *Seventh International Workshop on Serverless Computing (WoSC7) 2021 (WoSC '21), December 6, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3493651.3493669

1 SERVERLESS APPLICATION DEVELOPMENT

Function-as-a-Service (FaaS) was introduced by major cloud providers starting in 2014 as archetypical serverless programming platform. While it has since been complemented by further approaches such as the language-agnostic event-triggered Container-as-a-Service (CaaS), it is still immensely popular with application software engineers. Its simplicity stems from the ability to create individual short-running and stateless functions with low programming effort, orchestrate them into a larger application and deliver elastically scalable functionality according to a fine-grained pay-per-use model without worrying about maintaining anything other than the application itself. It is therefore considered an evolutionary advancement of the early Platform-as-a-Service (PaaS) models that, while also focusing on code, omitted some of the compelling execution characteristics. Under the hood, typically out of eyesight for application software engineers, FaaS originally relied on Docker containers but has since seen diversification, ranging from µ-VMs in AWS Lambda to zero-coldstart instances in Cloudflare Workers.

To support the creation of FaaS-based applications with conventional software development tools, code-level and runtime-level FaaSification approaches emerged. Investigation into FaaSification approaches has recently seen a surge in popularity and has consequently led to stream of FaaSifier tools as practical output. Zappa converts Python web applications into scalable AWS Lambda equivalents [13]. It includes a keep-warm functionality to avoid unresponsive websites due to cold starts. Aiming at data analytics and parallel job execution, PyWren was introduced to let non-experts harness the power of cloud computing [12]. It has since been picked up by IBM and evolved first into IBM PyWren and eventually into the Lithops framework [16]. Further approaches from the academic community include Termite and Nimbus for Java [3, 7], Lambada for Python [17], as well as Node2FaaS and DAF for JavaScript [6, 15]. In both Lambada and Termite, annotations (or decorators in Python terms) of the simple form `@CloudFunction` or the more instructive form `@CloudFunction(memory, duration, region)` control the code transpilation process to some degree. They cause the annotated methods, along with dependencies, to be packaged as cloud functions, and to be deployed and invoked on demand in a FaaS context. In DAF, annotations are available to indicate JavaScript (Node.js) package dependencies.

While the information attached to these annotations suffices to generate the cloud function configuration and packaging for the respective provider, the application software engineer is only seemingly detached from the infrastructure. As soon as the use of cloud functions exceeds trivial use cases, the engineer will have to interface with the FaaS control plane or even further cloud services in the stateful backend and infrastructure (BaaS, IaaS). This primarily applies to runtime patterns that have emerged over time as best practices [9]. For instance, containerised cloud functions are known to suffer from cold starts characterised by latency spikes [4]. In the absence of provider interfaces to specify latency requirements, a well-known orchestration pattern is to keep the function instance warm by regularly pinging it. This requires deploying an auxiliary function, connecting it to the main function, and setting up a timer trigger. The additional effort reduces the simplicity in serverless application development. Similar workaround patterns emerge when

WoSC '21, December 6, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-9172-6/21/12…$15.00
https://doi.org/10.1145/3493651.3493669

aiming at function instrumentation to facilitate debugging, when profiling the function memory consumption to properly size the instances, and when caching or persisting data. Moreover, several researchers have pointed out the rather un-simplistic effort to build FaaS-based applications with thousands of concurrent processes [8]. Instead of being an edge case, massive messaging needs from citizens and sensors in the ongoing digital transformation of societies will increase the demand for such applications whose setup complexity on the infrastructural level could be well wrapped within more powerful source code annotations. Hence, digital transformation can be shaped by a broader base of software engineers who focus on innovative applications instead of dealing with all syntactic details of cloud infrastructure orchestration.

We claim that our FaaS Fusion approach is the first to make such infrastructural patterns accessible on the code level as function annotations. Moreover, we postulate that next-generation cloud software engineering tools need to provide such pattern support to increase developer attractiveness and productivity. Patterns contribute to increasing the reuse potential in alignment with modular and service-oriented architectures due to the ability to hide the implementation [9]. In other words, engineers can focus on what should be achieved, less on how it should be achieved, and they can compare their work to best practices on an explicit and unambiguous level [5].

In the next section, we present the approach to capture patterns in annotations that are fitted into a code transpilation pipeline. Afterwards, we demonstrate a working implementation for JavaScript called FaaS Fusion. The implementation can be considered a meta-FaaSifier in that it offers an annotations library and corresponding transpiler framework based on the cross-platform Serverless Framework that can be further integrated into next-generation FaaSifiers. The remaining sections cover experimental evaluation results and the conclusion.

2 ANNOTATION TRANSPILATION APPROACH

2.1 Serverless Annotations and Patterns

Our work primarily targets the mechanics to wrap infrastructural concerns and function execution patterns in annotations. The space of patterns is rather large, unexplored and not formalised. We therefore investigated twelve exemplary patterns that have emerged from various serverless applications designs. We assume our approach can capture further patterns as long as the pattern scope can be determined to apply to a single variable or attribute (V), single function or method (F), helper functions or methods within an exposed function (H), or composite code unit (i.e., file or class) with set of functions or methods (U).

Table 1 gives an overview about the studied patterns, each of which is captured by a specific function annotation. Where references are given, the pattern itself is described in the existing literature along with a possible runtime mechanisms to implement the pattern in a cloud environment. However, annotations are intended to be reusable and therefore may lead to different pattern implementations. For instance, `@Autotune` could perform a one-time adjustment or continuous adjustment of the amount of memory allocated to a function under the control of the application, either within a fixed corridor or following the actual consumption, or it could entirely switch to cloud-managed services such as AWS Compute Optimizer, as long as the key goal – freeing the application engineer from having to specify a concrete amount of memory – is maintained [2, 18].

Table 1: Annotations, scopes and infrastructural patterns

Annotation	Scope	Pattern
`@CloudFunction`	F	Offload method invocation to cloud function in FaaS.
`@SecureFunction`	F	Offload method invocation to trusted execution environment.
`@LiquidFunction`	F	Offload method invocation dynamically to edge-cloud continuum.
`@Warmup` [19]	F	Minimise risk of function invocation with coldstart.
`@Autotune` [2]	F	Adjust function memory configuration to actual consumption.
`@Schedule`	F	Invoke function periodically based on time trigger.
`@HttpApi` [1]	F	Integrate function into web through exposed endpoint.
`@DeepTrace`	F	Activate tracing and input dependency detection.
`@Freshen` [10]	H	Install pre-execution and other lifecycle hooks.
`@Cache`	V	Ensure global data lifetime across invocations.
`@Persist`	V	BaaS (object store, database) data storage and retrieval.
`@CloudObject` [17]	U	Offload all methods and persist all attributes of class.

The annotations can have *inter-dependencies* with one implicitly requiring another. They can also be *compositional*, an important characteristic to raise the level of abstraction. The `@CloudObject` annotation would for instance instruct the FaaSification process to turn all methods of the annotated class definition into cloud functions (as if annotated with `@CloudFunction`) and turn all class attributes into persisted records in a BaaS store such as Redis, Memcached or S3 (as if annotated with `@Persist`). Finally, annotations are subject to either obligatory or optional *parameterisation*. For instance, the `@HttpApi` annotation requires HTTP method and relative URL path specifications, as in `@HttpApi(method=GET, path=/)`.

2.2 Annotations Syntax

While we refer to annotations on an abstract level, their addition to function code is subject to language-specific notations and constraints despite similar appearance in the `@`-notation. In Java, native annotations have been available since JDK 1.5/1.6. In Python, the term annotations refers to type hints in method declarations, whereas annotations as described above have also been available in native syntax since Python 2.4, via the addition of classes, under the name decorators. In JavaScript, there is no native support, although extensible compiler frameworks exist that lead to conventions of

including annotations as comments on the targets to be annotated. In general, their syntax can be described by the following rule:

```
[<COMMENT-SIGN>] @ANNOTATION-NAME [(PARAMKEY=
PARAMVALUE, ...)] [... COMMENT]
```

2.3 Transpilation Mechanics

The goal of the approach is to remain in line with the prevalent *serverless mindset*, shielding developers as much as possible from infrastructural concerns, while permitting to exploit common patterns involving functions on the FaaS level and backend functionality on the BaaS level. We therefore introduce the concept of *annotation bundles*, a library of annotations, each of which is associated with a desired runtime characteristic of an individual function and the corresponding modifications necessary to achieve those. They encompass runtime code modifications at the beginning of or in parallel to the function execution, function configuration changes, as well as additional deployments of backend services in the vicinity of a function.

Code transpilation takes higher-level code with annotations as references to annotation bundles and transforms it into a representation targeting a concrete environment. While there are techniques to rewrite code at runtime (dynamic optimisation [11] or monkey patching), our approach assumes the continued use of unaltered deployment and runtime tools to increase developer acceptance. It thus relies on static code transpilation that is injected into the build and deployment process of an application. Based on the Abstract Syntax Tree (AST) representation, annotations on all scopes (variable declarations, function definitions, classes) are extracted, matched against registered annotation bundles, and used to control the generation of a rewritten AST that furthermore adheres to the function invocation syntax of the respective language-provider combination. Fig. 1 summarises the underlying transpilation process.

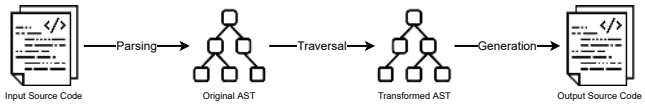

Figure 1: AST traversal based code transpilation

3 IMPLEMENTATION

We have implemented a meta-FaaSifier called FaaS Fusion to evaluate the feasibility and characteristics of code transpilation based on annotation bundles. As a proof of concept meta-tool, it is not meant to be used by software developers directly, but rather by researchers and developers who aim at creating the next generation of cloud application development tools including support for advanced FaaSification.

FaaS Fusion limits the previously introduced broad concepts in multiple ways to focus on an in-depth evaluation. First, it is implemented in JavaScript and only supports JavaScript code. Second, in its current version only AWS Lambda is supported as FaaS target environment, with few selected backend services on the infrastructure level. They encompass the AWS services API Gateway, CloudWatch, ElastiCache (Redis, Memcached) and S3. Third, only selected annotations are fully implemented - apart from the basic `@CloudFunction`, these are `@Warmup` and `@Autotune`, as well as `@HttpApi`. We allege that this subset is sufficient to validate the concept of annotations representing infrastructural patterns and provide the FaaS Fusion implementation as open source framework so that further annotations and infrastructural patterns can be investigated.

Fig. 2 refers to the FaaS Fusion workflow. A software engineer creates JavaScript projects with annotations added to code files. Next, the engineer runs the build/deploy instructions. FaaS Fusion is implemented as a transpilation plugin to the extensible JavaScript Babel compiler [14] that is invoked with reference to just the source and target directories. By running as part of Babel, FaaS Fusion is thus performing the code transpilation, as well as the generation of configuration serving as input to the Serverless Framework for deploying the functions into FaaS environments.

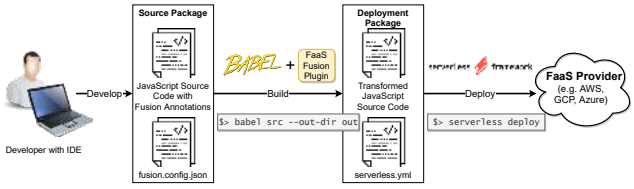

Figure 2: FaaS Fusion approach embedded into software engineering workflow

As additional benefit, Babel is able to backport modern JavaScript syntax into older language versions, thus increasing compatibility with FaaS providers that otherwise limit software developers to specific, and often slightly outdated, versions. For instance, in mid-2021, AWS Lambda supports Node.js 10, 12 and 14 (LTS), based on the V8 JavaScript engine in version 8, while 16.3 is the up to date version, based on V8 in version 9 with improved support for JavaScript (ECMAScript) language features such as match indices in regular expressions. Thus, many JavaScript application engineers are already used to running Babel as part of their workflows and will transparently gain support for FaaS annotations.

The complete annotations processing workflow is shown in greater detail in Fig. 3. Extracted variable and function annotations are matched against annotation bundles that are in turn parameterised with input from provider-specific configuration. The annotation bundles register a number of visitor methods that ensure all transpilation steps within a method at the code level, as well as necessary pre- and post-invocation methods, are occurring in the correct order. Eventually, the rewritten JavaScript code output, matching the language runtime of the chosen FaaS provider, is generated along with the YAML configuration file for the Serverless Framework. Hence, a sequence of running `babel` followed by `serverless deploy` results in ready-to-use Lambda functions with the desired characteristics as expressed in the annotations.

For each annotation, the abstract workflow is instantiated in concrete form by loading and running the respective visitor methods over the AST. Fig. 4 shows the instantiation for the `@Warmup` annotation that leads to the deployment of scheduled CloudWatch events, firing every 5 minutes with an event marked as warmup event. The

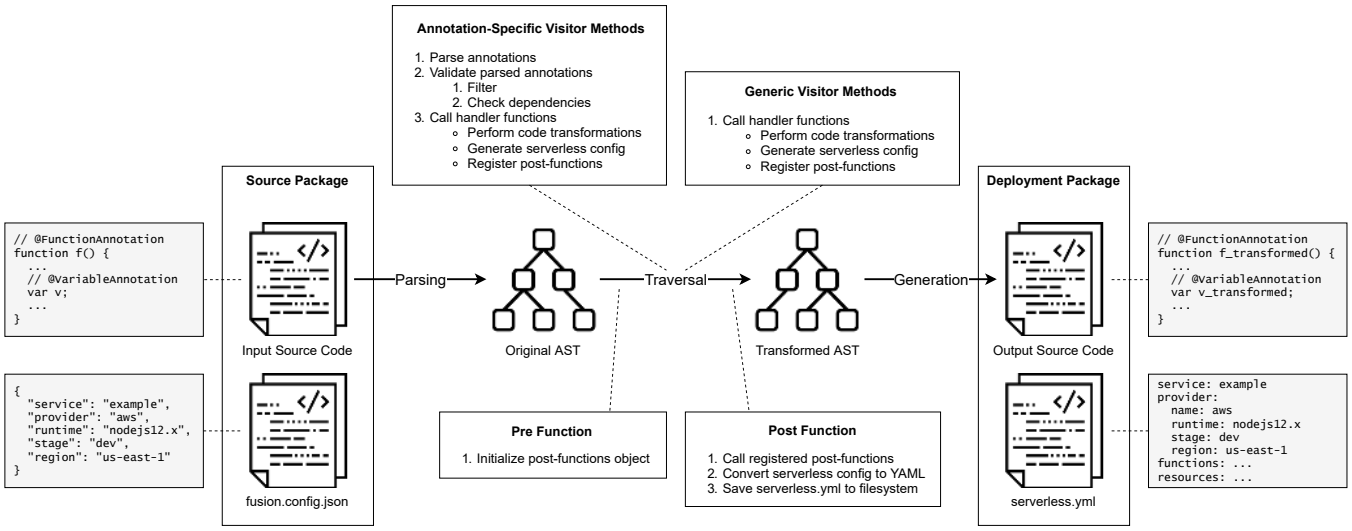

Figure 3: Internal annotations processing workflow based on AST traversal and provider-specific annotations bundles

function itself is rewritten to contain a conditional warmup event processing and early termination at the beginning.

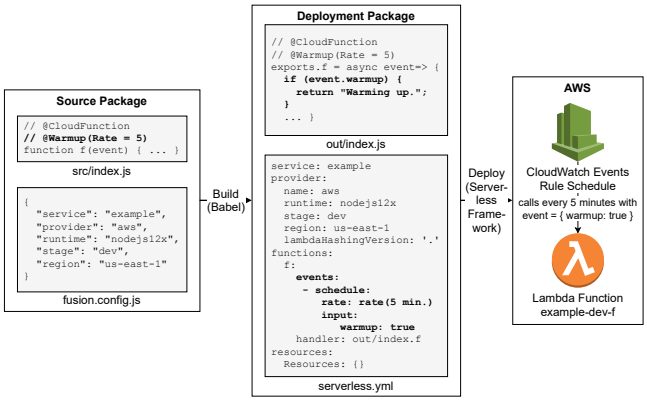

Figure 4: `@Warmup` annotation projected onto AWS Lambda and CloudWatch services

It should be noted that since 2019, Lambda has supported provisioned function concurrency that may on first sight reduce the need for `@Warmup` entirely. However, that comes at additional configuration effort and runtime cost. Such evolving implementation details are best shielded from the engineer through annotations, and `@Warmup` could be reused to address the evolution by another transpilation run that will switch to another implementation if deemed necessary.

The existing annotation implementations that target AWS Lambda perform cost-effective CloudWatch-scheduled function pinging (`@Warmup`), adjustment of function memory within a valid corridor according to the algorithm by Akhtar et al. (`@Autotune`), and automated registration of the function behind an API Gateway to make it accessible for RESTful invocation (`@HttpApi`).

4 EVALUATION RESULTS

4.1 Overview

In the following, we validate the feasibility of encapsulating infrastructural concerns in FaaS annotations. First, we evaluate the development process with FaaS Fusion, in particular its transpilation performance. Second, we observe the runtime behaviour of the resulting FaaS-deployed applications to assess whether the annotations have the right expressivity to yield noticeable advantages compared to omitting the annotations. The `@HttpApi` call is only validated qualitatively – does it work correctly or not – as there are no measurable metrics associated to its use other than measuring the registration time in the API Gateway itself, the slowness of which is already covered by the literature.

4.2 Development Process

While simplicity is a goal for application software engineers and raised code abstractions through powerful annotations are a means to accomplish that, it should not be traded off with unnecessary slowdown in the development cycle and especially the automated build process. We measured the transpilation performance with a representative engineer test system, a workstation with Intel i7-9700K octocore CPU clocked at 3.6 GHz and 16 GB RAM.

Fig. 5 shows the influence of including FaaS Fusion and the hosting Babel compiler into the build process depending on the size of a JavaScript file, measured over synthetically generated files with typical structures and complexity. One third of functions, each averaging 10 SLOC, is annotated with `@Warmup`, another third with `Autotune`. All functions are implicitly or explicitly marked as `@CloudFunction`.

The respective left bars establish the baseline by observing a pure Babel transpilation. For an otherwise interpreted or just-in-time compiled language such as JavaScript, this introduces noticeable overhead, but as previously described that is factored into the workflow of many application engineers already. The respective middle

bars show the combination of Babel and an annotation-less use of FaaS Fusion. They imply an almost constant build time addition amounting to around 30 ms on the test system, and thus a decreasing percentage of added build time of between 13% and 5% for the chosen file sizes. The respective right bars represent the inclusion and processing of annotations, with an increase of around 37% to 84% for larger files.

The result suggests that while modestly sized projects can benefit from the approach without noticeable overhead beyond the baseline, additional AST traversal optimisations such as AST caches are due to enable widespread use in complex code with many annotations.

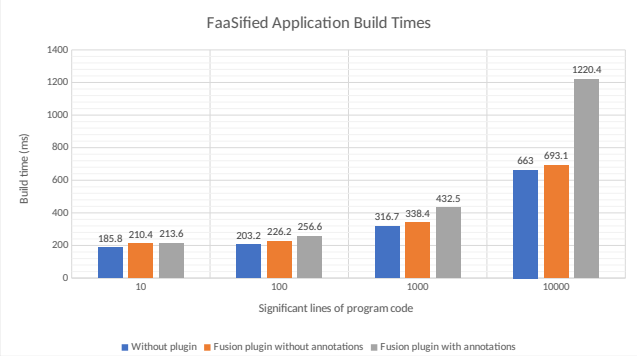

Figure 5: Influence of transpilation and FaaSification on the serverless application build time

4.3 Runtime Behaviour

Next, the behaviour of the `@Autotune` annotation for memory autotuning is evaluated.

Fig. 6 shows a 24-hour window over which the memory requirements of an exemplary function fluctuates, represented by bars. The statically assigned hull curve then shows the corresponding memory configuration of the function, including evidence of adhering to the configured minimum configuration around 03:17, 13:00 and at other times the actual consumption is much lower. A savings comparison is non-trivial due to the slowdowns inevitably caused by the reduced memory footprint. Assuming a function that includes wait times, the theoretic maximum saving is calculated based on the cost differences between the instance types that is proportional to the memory assignment Hence, for the scenario in Fig. 6 relative to a full 2048 MB function instance the theoretic maximum savings over one day, influenced by the choice of allocation corridor, is 60%.

Next, the behaviour of the `@Warmup` annotation for keeping functions warm is evaluated.

Fig. 7 first shows the function behaviour without regular warmup over a 72-hour window. The lower portions of the bar represent the function runtime, whereas the upper portions convey the overhead required for their initialisation that on average accounts for 59% and occasionally even exceeds the runtime (>100%). Reducing this overhead is the optimisation objective.

In contrast, Fig. 8 summarises the behaviour with warmup enabled over another 72-hour window. Warmup pings are sent in five minute intervals to 1 GB Lambdas executing in the AWS us-east

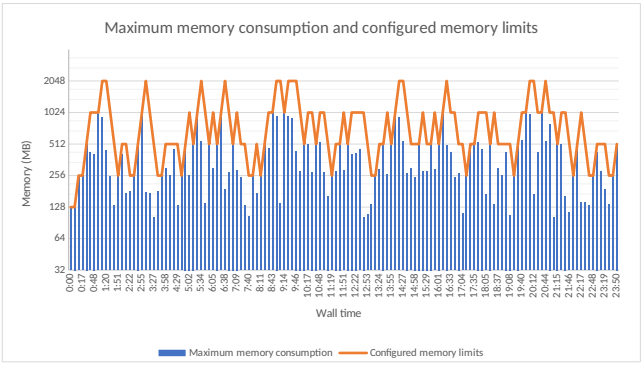

Figure 6: Effect of the `@Autotune` annotation on function memory allocation over time

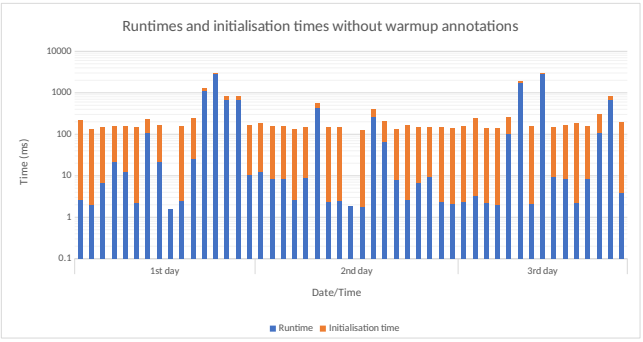

Figure 7: Initialisation and execution times, without `@Warmup` and thus with many coldstart situations

region. Their effective runtime is 1.28 ms, rounded up to 2 ms by Lambda's invoicing, resulting in effective warmup overhead of US$ 0.0000672 per day, or 0.002016 per month.

The initialisation overhead is reduced to only 2%, with a few occurrences of unexpected coldstart still happening due to the provider-enforced recycling of the function's underlying container. The savings exceed the additional cost as sum of the warmup calls and the overhead by far due to their short and constant runtime. The additional cost amount is so small that the AWS Calculator will budget it as US$ 0.00. Complementary, CloudWatch events cost US$ 1.00 per million events, a negligible factor for the expected 8640 pings per function instance per month adding less than US$ 0.01 to the invoice. Hence, adding warmup annotations according to our proposal represent a trivial and economic way to avoid coldstarts for functions that need it.

The evaluation has only covered single instances of functions. The effects on highly concurrent function invocation, along with opportunities to introduce additional annotations, remains an open research question.

5 CONCLUSIONS

Code annotations are a powerful means to simplify infrastructural concerns and patterns for developers of FaaS-based applications. We

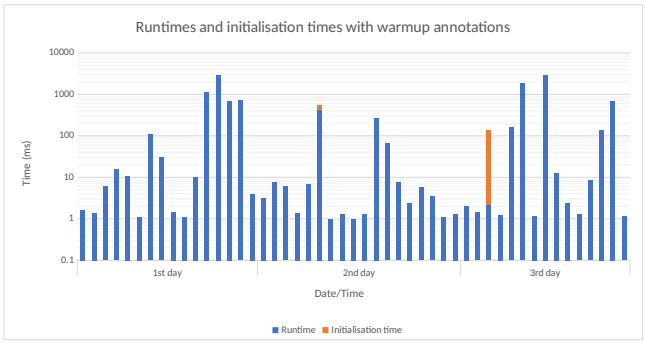

Figure 8: With `@Warmup`, almost no occurrence of initialisation time in addition to execution time

have proposed and investigated twelve annotations, partly based on mechanisms described by prior research.

To evaluate the annotations approach, we have implemented FaaS Fusion, a JavaScript meta-FaaSifier that currently provides annotation bundles for AWS Lambda, and relies on the Serverless Framework to facilitate cross-provider deployment of transpiled FaaS code in the future. FaaS Fusion currently handles four of the twelve annotations and is publicly available as open source tool[1].

On the research roadmap, we foresee the following four next steps. First, along with ongoing progress reported by researchers and practitioners, more patterns should be captured and implemented in FaaS Fusion as well as in existing FaaSification tools. Second, while we have implemented and validated the annotation bundles for the combination of JavaScript and AWS Lambda, we have not yet investigated other languages and other FaaS providers such as Google Cloud Functions, Azure Functions, IBM Cloud Functions or even different FaaS isolation models like Cloudflare Workers. This investigation should be conducted, for instance to study the portability of persistent variables marked with `@Persist` across providers. Third, optimised AST processing needs to be considered for large application projects with tens of thousands of lines of code. This can be accomplished by hooking into the caching mechanisms of modern compilers and transpilers, similar to the AspectJ weaving conducted by Termite. Fourth, while annotations are declarative, their current handling is imperative. Context-dependent smartness should be studied so that annotations that are not suitable could be overridden by system intelligence, their implementation paths could be chosen to yield optimal results, and similarly, annotations could be added automatically when there is an unambiguous advantage. This mechanism could also be used to balance optimisation objectives, such as regularly pinging function instances to reduce latency versus using provider-specific mechanisms like provisioned concurrency (for `@Warmup`), or self-determining the memory allocation needs versus using provider-specific mechanisms such as Compute Optimizer (for `@Autotune`).

A longer-term open research problem is the balance among multiple objectives, including green and sustainable serverless computing, secure serverless computing and other flavours.

[1]FaaS Fusion code: https://github.com/serviceprototypinglab/faasfusion

REFERENCES

[1] Gojko Adzic. 2018. Claudia.js Web API Example Project. online: https://github.com/claudiajs/example-projects/tree/master/web-api.

[2] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: Configuring Serverless Functions using Statistical Learning. In *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*. IEEE, 129–138. https://doi.org/10.1109/INFOCOM41043.2020.9155363

[3] Thomas Allerton. 2019. Nimbus – A framework for deploying and testing Java serverless applications. online: https://www.nimbusframework.com/.

[4] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. 2021. *Reducing Cold Starts during Elastic Scaling of Containers in Kubernetes*. Association for Computing Machinery, New York, NY, USA, 60–68. https://doi.org/10.1145/3412841.3441887

[5] Hayet Brabra, Achraf Mtibaa, Fábio Petrillo, Philippe Merle, Layth Sliman, Naouel Moha, Walid Gaaloul, Yann-Gaël Guéhéneuc, Boualem Benatallah, and Faïez Gargouri. 2019. On semantic detection of cloud API (anti)patterns. *Inf. Softw. Technol.* 107 (2019), 65–82. https://doi.org/10.1016/j.infsof.2018.10.012

[6] L. Carvalho and Aleteia P. F. de Araujo. 2020. Remote Procedure Call Approach using the Node2FaaS Framework with Terraform for Function as a Service. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*. INSTICC, SciTePress, 312–319. https://doi.org/10.5220/0009381503120319

[7] Serhii Dorodko and Josef Spillner. 2019. Selective Java code transformation into AWS Lambda functions. In *Proceedings of the European Symposium on Serverless Computing and Applications (CEUR-WS, Vol. 2330)*. 9–17.

[8] Sadjad Fouladi, Francisco Romero, Dan Iter, Quian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX'19)*. USENIX, Renton, WA, USA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[9] Sanghyun Hong, Abhinav Srivastava, William Shambrook, and Tudor Dumitras. 2018. Go Serverless: Securing Cloud via Serverless Design Patterns. In *10th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2018, Boston, MA, USA, July 9, 2018*, Ganesh Ananthanarayanan and Indranil Gupta (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud18/presentation/hong

[10] Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, and Eric Rozner. 2020. Proactive Serverless Function Resource Management. In *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. ACM, 61–66. https://doi.org/10.1145/3429880.3430102

[11] Animesh Jain, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. 2016. Continuous shape shifting: Enabling loop co-optimization via near-free dynamic code rewriting. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 23:1–23:12. https://doi.org/10.1109/MICRO.2016.7783726

[12] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 445–451. https://doi.org/10.1145/3127479.3128601

[13] Rich Jones. 2020. Zappa - Serverless Python. online: https://github.com/Miserlou/Zappa.

[14] Brian Ng, Henry Zhu, Huáng Jùnliàng, Logan Smyth, Nicolò Ribaudo, and Sven Sauleau. 2021. Babel – The compiler for next generation JavaScript. online: https://babeljs.io/.

[15] S. Ristov, S. Pedratscher, J. Wallnoefer, and T. Fahringer. 2021. DAF: Dependency-Aware FaaSifier for Node.js Monolithic Applications. *IEEE Software* 38, 1 (2021), 48–53. https://doi.org/10.1109/MS.2020.3018334

[16] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Software* 38, 01 (jan 2021), 68–74. https://doi.org/10.1109/MS.2020.3029994

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

[18] Josef Spillner. 2020. Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning. In *6th International Workshop on Serverless Computing (WoSC) / 21st ACM/IFIP Middleware*. online.

[19] Erwin van Eyk. 2018. Serverless Performance on a Budget. European Symposium on Serverless Computing and Applications (ESSCA) – http://essca2018.servicelaboratory.ch/slides/essca18-slides-erwinvaneyk-scaled.pdf.