A Framework for Bridging the Gap between Monolithic and Serverless Programming

Giuseppe De Palma[1] · Saverio Giallorenzo[1,2] · Matteo Trentin[1] · Gejsi Vjerdha[1]

[1] Alma Mater Studiorum — Università di Bologna, Bologna, Italy
[2] INRIA, Sophia-Antipolis, France

✉ gejsi.vjerdha@studio.unibo.it

1 Introduction

The state of the art of Cloud architectures is mainly divided between two styles. Microservices [11] see stateful processes expose multiple operations to users and developers manage both the provision of the servers running them and their scaling according to traffic fluctuations. Serverless functions (FaaS) [15] let developers deploy architectures made of stateless functions, each implementing a single operation; developers of serverless applications delegate the scaling of their architectures and the management of servers to the serverless platform.

In this paper, we focus on the latter style and present a framework that supports programmers in building serverless applications.

Indeed, the serverless programming model sees developers implement a distributed architecture out of independent functions. While this approach can work for simple coordination behaviours, growing the number of functions and interactions determines a "state explosion" which programmers can hardly reason upon to check the correctness of the implementation against its expected logic.

This experience is similar, although less fragmented, for the programmers of microservices, which concentrate on the development and integration of functionalities related to the same, coherent business logic provided by the operations exposed by the microservice. In microservices, the complexity of coordination emerges when multiple microservices need to interact in a distributed way—i.e., without a central *orchestrator*—so that the overall logic emerges from the interactions of the microservices. This second approach is usually called a *choreography* [20].

The most "linear" experience follows a monolithic development process, where programmers build their applications out of functions, which are components callable from other parts of the same application. Both static reasoning on the code and following the steps of execution is much simpler in this context, especially when using single-threaded models such as that of JavaScript. Moreover, generations of practitioners and researchers improved and honed the experience of programming software monoliths, providing guidelines, best practices, and tools suited for each phase of the development life cycle (design, programming, debugging, maintenance, etc.). Serverless, still in its infancy, is in dire need of such tools, which will likely take another generation to establish and reach mainstream adoption.

In these respects, the paradigm of choreographic programming [8, 9, 13, 18] uses choreographies as "monolithic" artefacts that specify the distributed logic of the system, relying on compilation to

© 2023 The Authors. This is document is shared under the terms of the Creative Commons Attribution License (CC-BY 4.0), which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

generate sets of components (e.g., connectors [7, 12]) that correctly implement the distributed logic of the system and mediate the interaction with the existing microservices.

In this research direction, we present the design and implementation of Fenrir, a programming framework that bridges the gap between monolithic and serverless programming. In Fenrir, developers write applications in a monolithic style. Then, they annotate which parts of the monolith shall be deployed as separate serverless functions, along with their respective call events (e.g., via external HTTP invocations, time-scheduled, etc.). Given the annotated codebase, the Fenrir compiler generates a correct-by-construction deployable serverless codebase following the annotations. Hence, Fenrir helps programmers achieve quick development and testing cycles, making sure that the execution semantics of the generated serverless application follows the one defined by the source program. Fenrir is available as an open-source project at `https://www.github.com/Gejsi/fenrir`. In Section 2, we present the main features of the Fenrir framework, namely, its annotation constructors and its compilation pipeline; we also briefly exemplify how Fenrir works in Section 2.1. We conclude by commenting on related and future work in Section 3.

2 The Fenrir Framework

As mentioned, Fenrir introduces annotations as an abstraction layer that the developers can unobtrusively use to apply code transformations and metadata generation to a given application, to deploy it on a serverless platform. Here, we focus on concrete annotations built for the popular AWS Lambda platform[4], but the concepts directly translate to similar serverless platforms, both private[3, 5] and open-source [1, 6, 14, 2].

Fenrir's annotations Fenrir relies on standard JSDoc comments, on which it introduces annotations as new special keywords that follow the pattern `/** $AnnotationName(param:"foo") */`. That pattern shows a crucial feature of Fenrir, i.e., users can *pass parameters to annotations*. This means that the user can customise how annotations define the translation process of a specific piece of the monolithic codebase. Besides primitive values (strings, numbers, etc.), annotation parameters are full-fledged JavaScript objects, such as arrays that carry multiple values or functions that specify custom behaviour used in the compilation process. Another important feature supported by Fenrir is the *composition* of annotations so that users can specify sequences of transformation steps, essentially defining dedicated compilation pipelines for each piece of the monolithic codebase.

Practically, each annotation corresponds to a code transformer, which is a visitor function that works on the annotated piece of source code to generate a modified version of it and/or related metadata. Core annotations supported by Fenrir are (we report their signature using TypeScript's syntax):

- `$Fixed(memorySize?: number, timeout?: number, ...)` converts monolithic functions into fixed-size serverless functions, whose resources are statically determined and remain constant regardless of the workload or input size. The annotation works by transforming the parameters and parts of the body of the functions (`return`/`throw` statements) to make them follow the platform's function signature (e.g., they are unary functions with an `event` parameter that carries the actual invocation parameters along with other runtime values);
- `$TrackMetrics(namespace: string, metricName: string, metricValue?: ts.Expression, ...)` generates code that monitors and logs the functions' resource usage—the annotation automat-

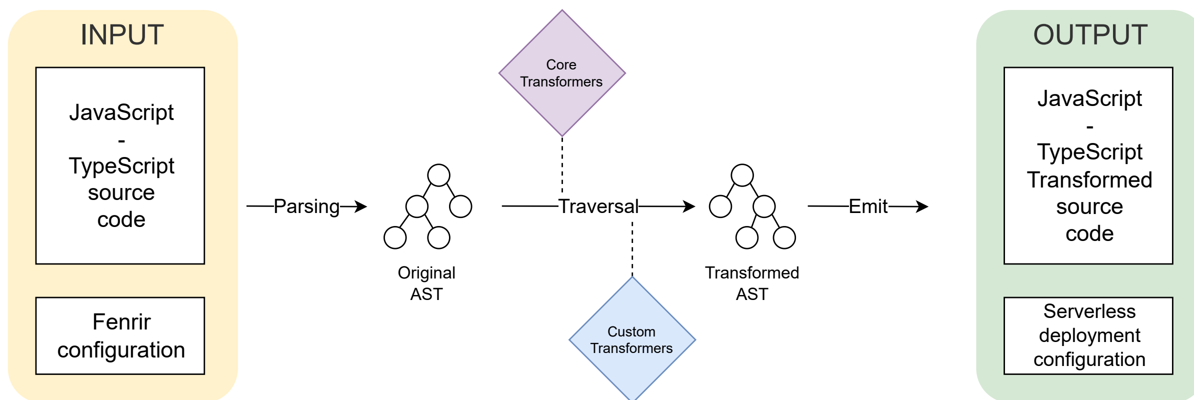

Figure 1: Fenrir's annotation-driven transpiler pipeline.

ically imports the necessary dependencies, e.g., for AWS it uses and injects the CloudWatch[1] dependency. The optional `metricValue` accepts any TypeScript expression, which is added to the function's body in a context-aware manner (e.g., if the expression feeds some data to a variable to monitor some measure, the monitoring code executes only after the expression);

- `$HttpApi(method: string, path: string, ...)` makes the function available at an HTTP endpoint through a set HTTP method;
- `$Scheduled(rate: string, ...)` makes the function run at specific dates or periodic intervals.

Besides the above annotations, Fenrir supports custom annotations, which let developers create their own transformers. Developers can publish their annotations/transformers and import them in a given codebase to assemble the compilation pipelines that best fit their deployment scenarios.

Fenrir's Compilation Process Technically, Fenrir is a transpiler that parses (user-written) annotations, builds the related pipeline of code transformers, and then processes each piece of annotated source code to generate its output. From the implementation standpoint, Fenrir performs the parsing and the transformations through the TypeScript compiler API, making the framework compatible with both TypeScript and JavaScript codebases—TypeScript codebases enjoy additional guarantees thanks to the type checker of the language, which Fenrir uses to also check user-defined transformers.

We complete our overview of Fenrir by looking at its transpilation process and pipeline, depicted in Figure 1. Starting from the left, after annotating their monolithic codebase, developers can use Fenrir's console interface to start the transpilation process. The tool provides step-by-step instructions to set the transpiler up (initialising the file `fenrir.config.json`) and handle the subsequent deployment.

The pipeline starts with the parsing of the input source code through the TypeScript compiler API, which produces AST nodes with their related annotations. Then, each annotation induces the application of its related transformation step, whose output is fed into the next transformer, if any. During the transformation steps, Fenrir reports possible errors by gracefully stopping the compilation process and indicating the offending instructions. Once the transformations have taken place without any errors, Fenrir's compiler saves the output source code, and it also appends the related metadata

[1] `https://aws.amazon.com/it/cloudwatch/`.

to a serverless.yml file—the latter specifies function deployment properties, e.g., the address to invoke a given function; specifically, the serverless.yml file makes the generated functions deployable through the Serverless framework[2].

2.1 From a Monolith to Serverless, by example

In Listing 1, we show an example of a monolithic codebase with a pair of illustrative functions. One function, called processOrder, retrieves orders (e.g., via a database query). The other function, called generateReport, produces reports based on the retrieved orders. Since we want the processOrder function to be invocable from clients, we annotate it as $Fixed, and we specify its HTTP endpoint and method with the $HttpApi annotation. The generateReport function is instead a backend one, which we want to run at pre-established intervals. To obtain this behaviour, we use the $scheduled annotation to specify that it shall be run every two hours.

Using Fenrir, we translate the code of Listing 1 into the serverless codebase of Listings 2– 4.

```
1  /**
2   * $Fixed
3   * $HttpApi(method: "GET", path: "/orders/report")
4   */
5  export async function processOrder(orderId) {
6    // ... processing logic ...
7    console.log(`Processing order ${orderId}`)
8    return order
9  }
10
11 /** $Scheduled(rate: "2 hours") */
12 export async function generateReport() {
13   // get the processed data and generate report
14   console.log("Generating report")
15 }
```

Listing 1: Source Code.

```
1 export async function processOrder(event) {
2   const orderId = event.orderId
3   // ... processing logic ...
4   console.log(`Processing order ${orderId}`)
5   return {
6     statusCode: 200,
7     body: JSON.stringify(order),
8   }
9 }
```

Listing 2: Generated Code, processOrder.

```
1 export async function generateReport() {
2  // get the processed data and generate report
3  console.log("Generating report")
4 }
```

Listing 3: Generated Code, generateReport.

```
1  processOrder:
2    handler: output.processOrder
3    events:
4      - httpApi:
5        method: GET
6        path: /orders/report
7  generateReport:
8    handler: output.generateReport
9    events:
10     - schedule:
11       rate: 2 hours
```

Listing 4: Generated Code, deployment configuration.

In Listings 2– 3, we find the respective processOrder and generateReport functions ready to be deployed on the serverless platform. In particular, notice that the input of processOrder changed to match the expected signature for functions of the serverless platform, i.e., an event that carries, among other content, the invocation parameters of the function, which are automatically assigned to local counterparts at the beginning of the function body. Complementarily, we also find the return value changed to match the shape of the response expected by the platform—at lines 5–8 of Listing 2, we create a JSON object with a status code and a body that contains a serialised version of the value held by the variable order, which holds the value returned by the function in the source codebase. The

[2] https://www.serverless.com/.

other notable element in the YAML code found in Listing 4, which contains the information that the serverless platform needs to deploy the two functions, e.g., the type of invocation for the `processOrder` function (HTTP) and its invocation address and the call schedule of the `generateReport` function.

3 Discussion and Conclusion

We presented Fenrir, a programming framework that aims to make the development of serverless applications as seamless as possible by letting developers write serverless architectures as traditional, monolithic programs. Fenrir's annotations let developers mark monolithic codebases to indicate what parts shall be deployed as serverless functions. Then, Fenrir's compiler applies annotation-induced transformations on the source code to generate an architecture amenable to serverless deployment. In doing so, Fenrir also promotes the incremental adoption of the serverless paradigm and supports developers in gradually learning serverless' deployment patterns.

Works closely related to Fenrir include similar tools that make a given codebase amenable to serverless deployment; so-called "FaaSifiers". The work closest to Fenrir are FaaSFusion, DAF, and M2FaaS [17, 22, 21]. The main difference between Fenrir and these proposals is in the objective behind the tools. Fenrir aims to build a serverless architecture starting from a monolithic codebase, which provides a more cohesive and responsive experience for developers, thanks to the consolidated techniques and set of tools available to programmers. The goal of FaaSifiers is that of offloading parts of the computation of a monolith to a serverless runtime, which is (intended to be) controlled and accessible only by the monolith itself. FaaSFusion and Fenrir are close also from the ergonomics standpoint since they block support function-level annotations. Contrarily, DAF and M2FaaS intersperse annotations within the code, to indicate which arbitrary lines of the monoliths shall become serverless units, including which values should be forwarded to functions, the dependencies that should be included, and which values should be returned to the monolith.

Another example is Node2FaaS [10], which is one of the earliest proposals in the field and, like Fenrir, targets JavaScript codebases. The main difference with Fenrir is that Node2FaaS deploys all functions of the monolith as separate serverless functions, providing no control over the many aspects of the deployment, like what functionalities are exposed by the serverless platform and which invocation modalities they accept (time-scheduled, via HTTP hooks).

Kallas et al. [16] recently presented mu2sls, a framework for transforming microservice applications into serverless ones; mu2sls uses a variant of Python with two extra primitives (transactions and asynchronous calls) to provide a formally-proven, correct-by-construction translation.

Future directions for Fenrir include the automatic support for closures (which one can implement as session-based calls to external databases) and the formalisation of the compilation process (formalising the annotations and transformations used by the compilation and, given some notion of behavioural correspondence, proving the correctness of the generated serverless code w.r.t. its source code), similar to the work conducted by Kallas et al. [16]. Moreover, we are interested in exploring how using choreographic languages, like Choral [13], can allow us to specify the interactions and behaviour of serverless functions. In particular, we conjecture that a choreographic language would allow us to express the patterns of interaction among the functions, e.g., supporting analyses such as finding the communication schemes that minimise the exchanges among the functions, to reduce the coordination overhead, and identifying/preventing possible anti-patterns, e.g., due to an under- or over-granularisation of the logic of functions—an anti-pattern seen also in microservices, called mega-/nano-services[19, 23].

References

[1] Apache openwhisk. https://openwhisk.apache.org/, 11 2022.

[2] Fission. https://fission.io/, 11 2022.

[3] Google cloud functions. https://cloud.google.com/functions/, 11 2022.

[4] Introducing aws lambda. https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/, 11 2022.

[5] Microsoft azure functions. https://azure.microsoft.com/, 11 2022.

[6] Openfaas. https://www.openfaas.com/, 11 2022.

[7] Marco Autili, Davide Di Ruscio, Amleto Di Salle, Paola Inverardi, and Massimo Tivoli. A model-based synthesis process for choreography realizability enforcement. In *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*, pages 37–52. Springer, 2013.

[8] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. *ACM SIGPLAN Notices*, 48(1):263–274, 2013.

[9] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13, 2017.

[10] Leonardo Rebouças de Carvalho and Aletéia Patrícia Favacho de Araújo. Framework node2faas: Automatic nodejs application converter for function as a service. In *CLOSER*, pages 271–278, 2019.

[11] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[12] Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*, pages 22–40. Springer, 2018.

[13] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases (pearl). In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[14] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[15] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[16] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. *Proceedings of the ACM on Programming Languages*, 7(POPL):367–395, 2023.

[17] Raffael Klingler, Nemanja Trifunovic, and Josef Spillner. Beyond @cloudfunction: Powerful code annotations to capture serverless runtime patterns. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, pages 23–28, 2021.

[18] Fabrizio Montesi. *Introduction to Choreographies.* Cambridge University Press, 2023.

[19] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems*, 35:3–15, 2020.

[20] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021.

[21] Stefan Pedratscher, Sasko Ristov, and Thomas Fahringer. M2faas: Transparent and fault tolerant faasification of node. js monolith code blocks. *Future Generation Computer Systems*, 135:57–71, 2022.

[22] Sasko Ristov, Stefan Pedratscher, Jakob Wallnoefer, and Thomas Fahringer. DAF: DependencyAware FaaSifier for Node.js Monolithic Applications. *IEEE Software*, 38(1):48–53, 2020.

[23] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–13, 2020.