Towards a Framework for Transitioning from Monolith to Serverless

Giuseppe De Palma[1,2], Saverio Giallorenzo[1,2], Jacopo Mauro[3](✉), Matteo Trentin[1,2,3], and Gejsi Vjerdha[1]

[1] Alma Mater Studiorum - Università di Bologna, Bologna, Italy
[2] OLAS team INRIA, Biot, France
[3] University of Southern Denmark, Odense, Denmark
mauro@imada.sdu.dk

Abstract. Serverless programming revolutionises the implementation of cloud architectures by allowing developers to deploy stateless functions without managing server infrastructure, enabling efficient scaling and resource usage. Serverless shifts to the cloud provider the burden of managing servers and scaling, enabling developers to focus solely on writing the code for the functionalities specific to a given architecture.

In this paper, we introduce Fenrir, a programming framework designed to facilitate the transition from monolithic programming to serverless. Fenrir enables developers to write applications in a monolithic style. Using annotation, users specify which components of the monolith shall implement separate serverless functions. Given these annotations, Fenrir generates a deployable serverless codebase, facilitating quick development and testing cycles while ensuring the alignment of the execution semantics between monolithic and serverless code.

1 Introduction

The landscape of Cloud architectures includes microservices [15] and serverless functions [23]. Each approach offers unique advantages and trade-offs regarding management and operational efficiency.

The microservices style advocates for the decomposition of applications into loosely-coupled services, each internally cohesive to encapsulate a specific business functionality—thus, microservices are usually "small" if compared to monolithic software that implements multiple, loosely-related functionalities. These stateful processes expose multiple operations to users and developers, affording granular control over individual components of an application. In the microservices paradigm, developers take responsibility for provisioning the servers that host these services and managing their scalability to accommodate fluctuations in user demand or traffic spikes. This approach necessitates a comprehensive understanding of the underlying infrastructure and entails proactive monitoring and adjustment of resources to maintain optimal performance.

Conversely, serverless functions, also called Function as a Service (FaaS), represent a departure from the traditional server-based approach to application

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
M. Hinchey and B. Steffen (Eds.): Festschrift Tiziana Margaria, LNCS 15240, pp. 167–182, 2025.
https://doi.org/10.1007/978-3-031-73887-6_13

deployment. In this paradigm, developers assemble a cloud application from the composition of stateless functions, each designed to execute a specific operation or task. Developers leverage cloud platforms to deploy these functions without the need to manage underlying servers or infrastructure explicitly. Indeed, programmers delegate the responsibility of scaling their architectures and managing server resources to the serverless platform provider. This abstraction of infrastructure management enables developers to focus exclusively on writing and deploying the code of functions, thereby streamlining the development process and reducing operational overhead. While microservices offer fine-grained control over resource allocation and performance optimisation, they may entail higher operational complexity and overhead due to the need for infrastructure management. Serverless architectures abstract away the complexities of infrastructure management, offering a pay-as-you-go model where the provider bills developers only for the resources consumed by their functions. However, this abstraction may introduce limitations on performance tuning and resource customisation, particularly for applications with stringent latency and resource requirements.

No matter what style is chosen, the growing number of functions or microservices found in cloud architectures causes an exponential explosion of the possible interactions a system can experience. This interaction explosion makes it hard for programmers to reason on the correctness of their implementations against their expected behaviour. In contrast, the experience of programming traditional monolithic software is much more "linear". In this style, programmers usually build their applications as a comprehensive codebase that includes all the logic into a single model. In this way, both static reasoning on the code and following the steps of execution are much simpler tasks than for the microservices and serverless cases.

Looking at recent advancements in the development of distributed systems, the paradigm of choreographic programming [10,13,18,32] uses choreographies as "monolithic" artifacts/models that specify the distributed logic of the system, relying on compilation to generate sets of components (e.g., connectors [7,17]). Like for model-driven engineering [24], the code automatically generated correctly implements the properties of the model (in this case, the distributed logic of the system) and possibly mediates the interaction with the existing components (e.g., microservices).

Inspired by choreographic programming and model-driven engineering, we aim to support programmers in building serverless applications. For this reason, in this paper, we present the design and implementation of Fenrir, a programming framework that facilitates the transition between monolithic and serverless programming. In Fenrir, developers write applications in a monolithic style. Then, they annotate which parts of the artefact 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, as it happens in choreographic programming, Fenrir 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 follow the one defined by the source program. Fenrir is available as an open-source project at https://www.github.com/Gejsi/fenrir.

Lessons Learned from Tiziana Margaria. We present this contribution in honor of Tiziana on the occasion of her 60th birthday. Tiziana Margaria's career has significantly shaped the field of programming safe and formally verified systems and her work has been instrumental in bridging the gap between theoretical foundations and practical applications in software engineering and many other domains such as healthcare [40], agriculture [20], and history [8]. Starting from her work in the telecommunication services [43] and later on Service-Oriented design [30] and Web-Service Construction [25], she always advocates for a "divide and conquer" approach where initial prototypes are successively modified until each component satisfies the requirements, paying attention to the fact that the generation of services is constantly accompanied by verification of the validity of the required features. Key to her approach is the notion of Model and Model-Driven Engineering [12,28] that allows at the same time to describe the behaviour of the program, reason on the correctness of the system, and generate the code that correctly implements it. Talking to her, it is easy to see that Tiziana is an energetic and keen believer in using Model-Driven Engineering to lower the entry barriers for software development, always recommending involving all the stakeholders in the design process [29] and more recently even pushing the boundaries with low-code/no-code approaches [11,34].

This proposal integrates some of Tiziana's ideas applied to the domain of cloud application development. Indeed, Fenir follows a Model-Driven development approach in which developers write a model of the system, delegating the generation of the code to a tool that exploits annotations to correctly implement architecture components on serverless platforms. We hope that this work can contribute to the lowering of the entry barriers for developing correct-by-constructions serverless applications.

Structure of the Paper. In Sect. 2, we start by introducing Fenrir using a running example. In Sect. 3, we present an overview of serverless programming. We introduce in Sect. 4 the main features of Fenrir, namely, its annotation constructors and its pipeline. In Sect. 5, we show how Fenrir works with a more elaborate example. We conclude by commenting on related and future work in Sect. 6.

2 Introductory Example

We start by introducing the experience of using Fenrir with a running example (expanded and explained in detail in Sect. 5) of a monolithic JavaScript codebase with a pair of illustrative functions that transform into a serverless architecture. In the codebase, one function, called `processOrd`, retrieves orders (e.g., via a database query). The other function, called `generateRep`, produces reports based on the retrieved orders.

```
1  export async function processOrd(orderId) {
2  // ... processing logic ...
3    return order
4  }
5  export async function generateRep() {
6    //... report generation logic ...
7  }
```

Given the code above, we introduce annotations for the `processOrd` and `generateRep` functions to make them separate serverless functions via the Fenrir code generator. Specifically, we annotate the first function to make it callable from clients via HTTP requests. The second is instead a backend batch function that shall run every two hours. We report below the except of the code above with the Fenrir annotations of the two functions.

```
1  /**
2  * $Fixed
3  * $HttpApi(method: "GET", path: "/orders/report")
4  */
5  export async function processOrd(orderId) {
6    ...
7  }
8  /** $Scheduled(rate: "2 hours") */
9  export async function generateRep() {
10   ...
11 }
```

Briefly, we annotate `processOrd` as `$Fixed` to mean it is a fixed-size serverless function—the fixed-size attribute means that the resources assigned to the function (e.g., CPU, RAM) are constant and statically determined regardless of the workload or input size—and that it must be exposed as an `$HttpApi` reachable through the GET HTTP method at the URL `/orders/report` (the annotation abstracts away the address of the server hosting the function, bound at deployment time). Similarly, we annotate `generateRep` to be `$Scheduled` at a `rate` of once every `"2 hours"`.

Concentrating, for brevity, on the result of the process of `processOrd` (the results for the other function are similar), we obtain two artefacts. The first is the JavaScript code of the serverless implementation of `processOrd`, reported below on the left. The most notable traits of the translation regard the transformation of the input of `processOrd` to match the expected signature for functions of the serverless platform, i.e., an event `e` 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, where 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 carried the returned output of the function in the source codebase. The second artefact generated by the code generator is the YAML code found on the right, which contains the information that the serverless platform needs to deploy the

function, i.e., the type of invocation (HTTP, with method and path) for the `processOrd` function.

```
1 export async function processOrd(e){
2   const orderId = e.orderId
3   // ... processing logic ...
4   return {
5     statusCode: 200,
6     body: JSON.stringify(order)
7   }
8 }
```

```
1 processOrd:
2 handler: output.
      processOrd
3 events:
4   - httpApi:
5       method: GET
6       path: /orders/
            report
```

3 Preliminaries

In this section, we provide a brief overview of serverless computing and the platforms that support it.

Modern cloud applications have access to a plethora of services that allow them to scale and be more resilient. As they can scale more, their complexity and usage increases, leading to the need to be efficiently (and automatically) managed. Serverless computing was born to respond to these needs by offering a kind of service that abstracts away the underlying infrastructure, and allows an application to be built as a composition of stateless, event-driven functions that can automatically scale up and down.

A serverless application is written via software units called functions, which are run in short-lived environments triggered by some kind of event. When a function invocation is triggered by an event such as HTTP requests, database changes, file uploads, scheduled intervals or various other triggers, the provider runs the code after initializing an execution environment, a secure and isolated context that manages all the resources needed for the function lifecycle. Execution environments are technically handled differently by the platform providers, e.g. Virtual Machines (VMs), μVMs or Docker containers, etc.

Figure 1 presents the typical architecture of a serverless platform. The main components are the controllers and the workers. The controller receives requests from external sources, such as users or other systems. It handles scaling decisions based on incoming traffic and system load, orchestrates the allocation of worker nodes for function execution and manages the overall system coordination and monitoring. The scheduler in particular determines which worker node should execute each function based on factors such as current load, function requirements, and resource availability. Worker nodes then execute the actual functions requested by the controller node, handling the execution environment lifecycle, including provisioning, scaling, and teardown.

Serverless platforms usually adopt a communication layer that facilitates communication between the controller node and worker nodes, handling messages and data transfer between components. In particular, message queues or event brokers (e.g., RabbitMQ [?], Kafka [5]) are used for asynchronous communication between components, allowing decoupling and scalability. Internal APIs

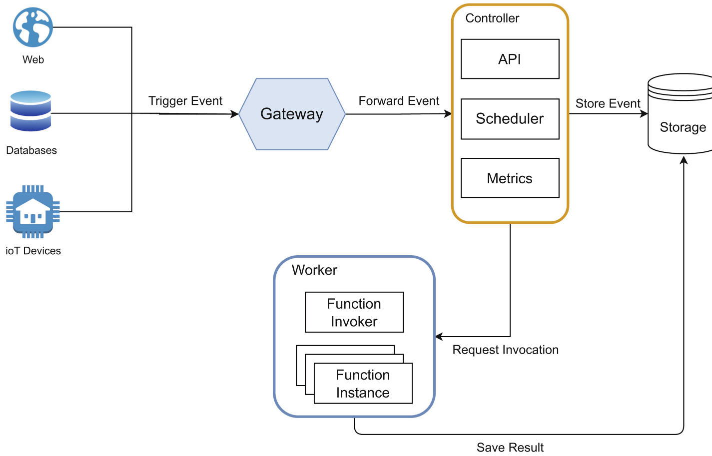

Fig. 1. Typical serverless platform architecture.

or RPC facilitate synchronous communication for tasks such as function deployment, status updates, and resource allocation. Monitoring tools are also used to collect metrics on resource usage, function execution times, and error rates. Metrics provide visibility into system performance, function execution, and overall health and enable debugging, troubleshooting, and performance optimization.

In serverless platforms, events play a crucial role in triggering function executions and driving the serverless architecture. Platforms often support a variety of events ranging from HTTP requests for handling webhooks and web-based interactions, cloud storage events (e.g., creation, deletion, or modification of an object in the cloud storage system or database activities such as inserts, updates, or deletions of records), events triggered at predefined intervals or specific times, events triggered by messages arriving in a message queue or streams, and events generated by custom sources or external systems via integrations or APIs.

Among the leading providers of serverless computing platforms, Amazon Web Services (AWS) Lambda [4] stands out as a pioneer in the field. AWS Lambda was the first publicly available serverless platform, allowing developers to pay only for the compute time consumed by their functions. Other platforms followed suit, offering similar capabilities, such as Microsoft Azure Cloud Functions [31] and Google Cloud Platform (GCP) Cloud Functions [19].

A number of open-source serverless platforms have also emerged, such as OpenWhisk [6], Knative [44], and OpenFaaS [35]. These platforms can be deployed on-premises or on the cloud, and offer a more flexible and customizable solution compared to the proprietary platforms.

4 The Fenrir Framework

The idea behind Fenrir is to use 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 [19,31] and open-source [6,16,21,35].

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 code generation process. Another important feature supported by Fenrir is the *composition* of annotations so that users can specify sequences of transformation steps, essentially defining 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):

- `$Fixed(memorySize?: number, timeout?: number, ...)` converts monolithic functions into fixed-size serverless functions, whose resources are statically determined and remain constant regardless of the workload or input size. The annotation works by transforming the parameters and parts of the body of the functions (`return`/`throw` statements) to make them follow the platform's function signature (e.g., they are unary functions with an `event` parameter that carries the actual invocation parameters along with other runtime values). Note that functions without the`$Fixed` annotation are considered local functions and are included in the body of other annotated functions.
- `$TrackMetrics(namespace: string, metricName: string, metricValue?: ts.Expression, ...)` generates code that monitors and logs the functions' resource usage—the annotation automatically imports the necessary dependencies, e.g., for AWS it uses and injects the CloudWatch [3]. dependency. The optional `metricValue` accepts any TypeScript expression, which is added to the function's body in a context-aware manner (e.g., if the expression feeds some data to a variable to monitor some measure, the monitoring code executes only after the expression);
- `$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 annota-

tions/transformers and import them in a given codebase to assemble the compilation pipelines that best fit their deployment scenarios.

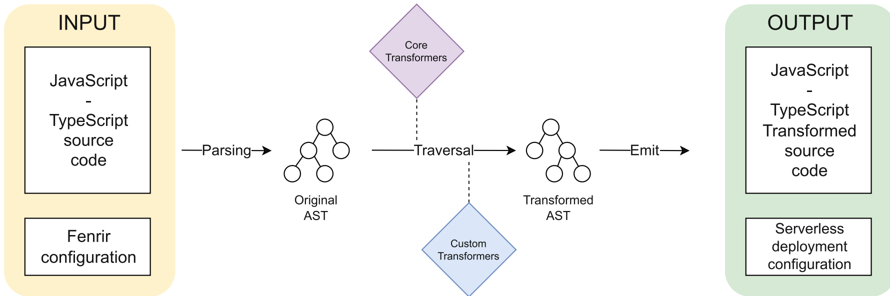

Fig. 2. Fenrir's annotation-driven pipeline.

Fenrir's Workflow. Fenrir 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 is also used to check user-defined transformers.

We complete our overview of Fenrir by looking at its pipeline, depicted in Fig. 2. Starting from the left, after annotating their monolithic codebase, developers can use Fenrir's console interface to start the code generation process. The tool provides step-by-step instructions to set up the code generation (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 process and indicating the offending instructions. Once the transformations have taken place without any errors, Fenrir saves the output source code, and it also appends the related metadata 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 [42].

5 From a Monolith to Serverless, by Example

To better visualise the developer's programming tasks, in this section, we exemplify how Fenrir could be used to transform an application written as a monolith into a serverless cloud application.

Let's consider a subset of an example e-commerce application. We want this application to, at least:

1. allow for the insertion of a new order, representing the purchase of an item by a specific user;
2. allow for the retrieval of information about an order, given its identifier;
3. generate a report of all processed orders periodically

Such functionalities would interact directly with an underlying storage (e.g. a database), to retrieve and insert the required orders. In Listing 1.1, we show these three basic features, in the form of functions taken from a monolithic codebase.

One function, called `insertOrder`, receives in input the ID of the product being purchased, its amount, and the ID of the user making the purchase; it then generates an ID for the corresponding order, stores the data in a database, and returns the ID to the caller. The `retrieveOrder` function performs the inverse operation: given the ID of an order, it retrieves the related information from storage, and returns it to the caller. Finally, the `generateReport` function produces reports based on the processed orders. Since we want both `insertOrder` and `retrieveOrder` to be invocable by clients and not treated as local functions, we annotate them as `$Fixed` , and we specify their HTTP endpoints and methods with the `$HttpApi` annotation (POST and GET respectively, reflecting their behaviour). The `generateReport` function, however, is a backend functionality that does not require (nor expect) user interaction; instead, we want it 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 1.1 into the serverless codebase of Listings 1.2 and 1.3.

In Listing 1.2, we find all three functions ready to be deployed on the serverless platform. In particular, notice that the input of both `insertOrder` and `retrieveOrder` 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 values changed to match the shape of the response expected by the platform—at lines 5–8 and 18–21 of Listing 1.2, we create a JSON object with a status code and a body that contains a serialised version of the value held by the return variable in the source codebase (i.e. `orderId` for `insertOrder`, `order` for `retrieveOrder`). As for `generateReport`, the function's body is unchanged, given that it doesn't take any inputs and doesn't return any outputs, so from Fenrir's point of view, it is comprised only of its internal logic. The other notable element is the YAML code found in Listing 1.3, which contains the information that the serverless platform needs to deploy the three functions, e.g., the type of invocation for the `processOrder` function (HTTP) and its invocation address and the call schedule of the `generateReport` function.

```
/**
 * $Fixed
 * $HttpApi(method: "GET", path: "/orders/retrieve")
 */
export async function retrieveOrder(orderId) {
  console.log(`Retrieving order ${orderId}`)
  // ... retrieve order from database ...
  return order
}

/**
 * $Fixed
 * $HttpApi(method: "POST", path: "/orders/insert")
 */
export async function insertOrder(productId, amount, userId) {
  // ... processing logic ...
  console.log(`Order ${orderId} inserted`)
  return orderId
}

/** $Scheduled(rate: "2 hours") */
export async function generateReport() {
  console.log("Generating report")
  // get the processed data and generate report
}
```

Listing 1.1. Source code.

6 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 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 [27,39,41]. 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

```
1  export async function retrieveOrder(event) {
2    const orderId = event.orderId
3    console.log(`Retrieving order ${orderId}`)
4    // ... processing logic ...
5    return {
6      statusCode: 200,
7      body: JSON.stringify(order), }}
8
9  export async function insertOrder(event) {
10   const productId = event.productId
11   const amount = event.amount
12   const userId = event.userId
13
14   // ... processing logic ...
15   console.log(`Order ${orderId} inserted`)
16   return {
17     statusCode: 200,
18     body: JSON.stringify(orderId) }}
19
20 export async function generateReport() {
21   // get the processed data and generate report
22   console.log("Generating report") }
```

Listing 1.2. Generated code.

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 [14], 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. [26] 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.

We deem Fenrir a valid prototype to showcase the promising approach of building serverless architectures out of a monolithic codebase. However, we see interesting future directions that pose challenges to both research and development for overcoming the limitations of the current implementation. Firstly, Fenrir cannot handle global mutable variables within functions' bodies or perform filesystem operations. In a monolithic approach, global variables and files

```
1  retrieveOrder:
2    handler: output.retrieveOrder
3    events:
4      - httpApi:
5        method: GET
6        path: /orders/retrieve
7  insertOrder:
8    handler: output.insertOrder
9    events:
10     - httpApi:
11       method: POST
12       path: /orders/insert
13 generateReport:
14   handler: output.generateReport
15   events:
16     - schedule:
17         rate: 2 hours
```

Listing 1.3. Generated code, deployment configuration.

can be used to store information such as the state of the application. However, since Fenrir transforms functions to be executed in a fully distributed system, it becomes impossible to use the filesystem or global state for storing information, as these are not available in a fully distributed environment. Developers can achieve the same functionalities by refactoring global mutable variables to be passed as parameters to functions and returned as results, avoiding the often-considered bad practice of using global variables [38]. Developers also need to translate operations using the file system to add calls to external storage services, as functions do not share memory (rendering mutable global variables unusable) or a common persistent filesystem.

While annotated functions can call non-annotated functions by including them as local function calls, another missing feature is that, in Fenrir, annotated functions cannot invoke other annotated functions. This cross-function calls restriction is to prevent an antipattern where serverless functions call each other through their respective endpoints. While in a monolithic approach this function-to-function call style is the norm—where any procedure call will simply add a frame on the stack in the server memory without big overheads—such calls would significantly increase latency in a serverless setting. The invoked functions would indeed need to pass through the serverless platform's entry point and load balancer repeatedly. New proposals of serverless frameworks such as the one by Jia and Witchel [22] are trying to reduce the overhead of cross-function calls, e.g., optimising internal function calls locally on the same worker without going through the API gateway, for the time being, we decided to enforce this constraint.

Related to the previous point, Fenrir does not support higher-order annotated functions. Implementing higher-order functions, i.e., functions that take other

functions as arguments or return them as results, in a distributed setting can be challenging because these functions often involve passing other functions as arguments or returning them as results. In particular, closures, which capture the local state of a function, are difficult to manage in a distributed environment because the captured state must be serialised and transferred across nodes. Hence, supporting this feature would require maintaining function references and states across different nodes in the distributed system, which can introduce significant complexity.

Additionally, Fenrir currently implements coarse-grained error handling: when the runtime detects an exception in an annotated function it transforms it into a 400 HTTP response, similar to how Fenrir transforms the returns found in the source code into 200 HTTP responses. We plan to consider a more fine-grained approach in the future, e.g., using annotations to refine what kind of HTTP error exceptions shall transform into.

Finally, looking at scheduling policies, Fenrir could provide insights on how much cold starts affect the serverless function due to its external dependencies. Cold starts can introduce latency because the serverless platform needs to initialise the function's runtime environment, which includes loading any required libraries and dependencies. Hence, developers must consider the size and complexity of their functions' dependencies to minimise the impact on deployment performance. Since Fenrir has (at least part) of the information needed to provide insight on this issue, future work can develop tools that, starting from Fenrir's annotations can help the developers in devising the architectural division and implement the optimisations needed to ensure that their functions will be lightweight and efficient.

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 annotations and transformations performed to prove the correctness of the generated serverless code w.r.t. its source code, similar to the work conducted by Kallas et al. [26]. Moreover, we are interested in exploring how using choreographic languages, like Choral [18], 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 antipatterns, e.g., due to an under- or over-granularisation of the logic of functions—an antipattern seen also in microservices, called mega-/nano-services [33,45].

Another interesting line of work regards the possibility of optimising the workflow of the generated FaaS application. Indeed, a recent trend of FaaS is the definition/handling of the composition/workflows of functions, like AWS step-functions [2] and Azure Durable functions [9]. The main idea behind these works is to allow users to define workflows as the composition of functions with their branching logic, parallel execution, and error handling. The orchestrator/-controller of the platform then uses the workflow to manage function executions

and handle retries, timeouts, and errors. Similarly, recent work [1,36,37] used optional opaque parameters or scheduling constraints in function invocations to inform the load balancer on the affinity with previous invocations and the data they produced or to control where the functions need to be run. Following this direction, one could extend Fenrir's annotation to incorporate also this information, thus allowing the final serverless application to be more efficient, robust, and reliable.

Conflict of Interest. The author(s) has no competing interests to declare that are relevant to the content of this manuscript.

References

1. Abdi, M., et al.: Palette load balancing: locality hints for serverless functions. In: Proceedings of the Eighteenth European Conference on Computer Systems, pp. 365–380 (2023)
2. Amazon Web Services. AWS Step Functions (2023). https://aws.amazon.com/step-functions/. Accessed July 2024
3. Amazon Web Services. Amazon CloudWatch (2024). https://aws.amazon.com/it/cloudwatch/. Accessed July 2024
4. Amazon Web Services. Introducing AWS Lambda (2024). https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/. Accessed July 2024
5. Apache Software Foundation. Apache Kafka (2024). https://kafka.apache.org/. Accessed July 2024
6. Apache Software Foundation. Apache OpenWhisk (2024). https://openwhisk.apache.org/. Accessed July 2024
7. Autili, M., Di Ruscio, D., Di Salle, A., Inverardi, P., Tivoli, M.: A model-based synthesis process for choreography realizability enforcement. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 37–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_4
8. Breathnach, C., Murphy, R., Schieweck, A., O'Shea, E., Clancy, S., Margaria, T.: Curating history datasets and training materials as OER: an experience. In: Shahriar, H., et al. (eds.) IEEE Annual Computers, Software, and Applications Conference, COMPSAC, pp. 1570–1575. IEEE (2023)
9. Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S.: Durable functions: semantics for stateful serverless. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021)
10. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. ACM SIGPLAN Not. **48**(1), 263–274 (2013)
11. Chaudhary, H., Margaria, T.: Integration of micro-services as components in modeling environments for low code development. In: Proceedings of the Institute for System Programming of the RAS (ISP RAS), vol. 33, pp. 19–30 (2021)
12. Chaudhary, H.A.A., et al.: Model-driven engineering in digital thread platforms: a practical use case and future challenges. In: Margaria, T., Steffen, B. (eds.) ISoLA 2022. LNCS, vol. 13704, pp. 195–207. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19762-8_14

13. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: theory and implementation. Log. Methods Comput. Sci. **13**, 1–57 (2017)
14. de Carvalho, L.R., de Araújo, A.P.F.: Framework Node2FaaS: automatic NodeJS application converter for function as a service. In: CLOSER, pp. 271–278 (2019)
15. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
16. Fission Project. Fission (2024). https://fission.io/. Accessed July 2024
17. Giallorenzo, S., Lanese, I., Russo, D.: ChIP: a choreographic integration process. In: Panetto, H., Debruyne, C., Proper, H.A., Ardagna, C.A., Roman, D., Meersman, R. (eds.) OTM 2018. LNCS, vol. 11230, pp. 22–40. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02671-4_2
18. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: the choreographic and multitier cases (pearl). In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP. LIPIcs, vol. 194, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
19. Google. Google Cloud Functions (2024). https://cloud.google.com/functions/. Accessed July 2024
20. Guevara, I., Ryan, S., Singh, A., Brandon, C., Margaria, T.: Edge IoT prototyping using model-driven representations: a use case for smart agriculture. Sensors **24**(2), 495 (2024)
21. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with OpenLambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2016) (2016)
22. Jia, Z., Witchel, E.: Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: Sherwood, T., Berger, E.D., Kozyrakis, C. (eds.) ASPLOS 2021: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 152–166. ACM (2021)
23. Jonas, E., et al.: Cloud programming simplified: a berkeley view on serverless computing. arXiv preprint arXiv:1902.03383 (2019)
24. Jörges, S., Margaria, T., Steffen, B.: Assuring property conformance of code generators via model checking. Formal Aspects Comput. **23**(5), 589–606 (2011)
25. Jung, G., Margaria, T., Nagel, R., Schubert, W., Steffen, B., Voigt, H.: SCA and jABC: bringing a service-oriented paradigm to web-service construction. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 139–154. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88479-8_11
26. Kallas, K., Zhang, H., Alur, R., Angel, S., Liu, V.: Executing microservice applications on serverless, correctly. Proc. ACM Program. Lang. **7**(POPL), 367–395 (2023)
27. Klingler, R., Trifunovic, N., Spillner, J.: Beyond @cloudfunction: powerful code annotations to capture serverless runtime patterns. In: Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021, pp. 23–28 (2021)
28. Margaria, T., Steffen, B.: Service-orientation: conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2297-5_10
29. Margaria, T., Steffen, B.: eXtreme Model-Driven Development (XMDD) technologies as a hands-on approach to software development without coding. In: Tat-

nall, A. (ed.) Encyclopedia of Education and Information Technologies, pp. 1–19. Springer, Cham (2020). https://doi.org/10.1007/978-3-319-60013-0_208-1
30. Margaria, T., Steffen, B., Reitenspieß, M.: Service-oriented design: the roots. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 450–464. Springer, Heidelberg (2005). https://doi.org/10.1007/11596141_34
31. Microsoft. Microsoft Azure Functions (2024). https://azure.microsoft.com/. Accessed July 2024
32. Montesi, F.: Introduction to Choreographies. Cambridge University Press, Cambridge (2023)
33. Neri, D., Soldani, J., Zimmermann, O., Brogi, A.: Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Softw.-Intensive Cyber-Phys. Syst. **35**, 3–15 (2020)
34. University of Limerick. Raise project (2024). https://software-engineering.ie/raise/. Accessed July 2024
35. OpenFaaS Ltd. OpenFaaS (2024). https://www.openfaas.com/. Accessed July 2024
36. Palma, G.D., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: A declarative approach to topology-aware serverless function-execution scheduling. In: IEEE International Conference on Web Services, ICWS, pp. 337–342. IEEE (2022)
37. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation priority policies for serverless function-execution scheduling optimisation. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) ICSOC 2020. LNCS, vol. 12571, pp. 416–430. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65310-1_29
38. Parkhe, R.: Global variables are bad (2024). https://wiki.c2.com/?GlobalVariablesAreBad. Accessed July 2024
39. Pedratscher, S., Ristov, S., Fahringer, T.: M2FaaS: transparent and fault tolerant FaaSification of Node.js monolith code blocks. Future Gener. Comput. Syst. **135**, 57–71 (2022)
40. Rehman, M., Javed, I.T., Qureshi, K.N., Margaria, T., Jeon, G.: A cyber secure medical management system by using blockchain. IEEE Trans. Comput. Soc. Syst. **10**(4), 2123–2136 (2023)
41. Ristov, S., Pedratscher, S., Wallnoefer, J., Fahringer, T.: DAF: Dependency-Aware FaaSifier for Node.js monolithic applications. IEEE Softw. **38**(1), 48–53 (2020)
42. Serverless Inc. Serverless (2024). https://www.serverless.com/. Accessed July 2024
43. Steffen, B., Margaria, T., Claßen, A., Braun, V., Nisius, R., Reitenspieß, M.: A constraint-oriented service creation environment. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 418–421. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_63
44. The Knative Authors. Knative (2024). https://knative.dev/docs/. Accessed July 2024
45. Tighilt, R., et al.: On the study of microservices antipatterns: a catalog proposal. In: Proceedings of the European Conference on Pattern Languages of Programs 2020, pp. 1–13 (2020)