ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Fenrir: A framework for enhancing serverless programming through annotation-driven transformations

Relatore:
Chiar.mo Prof.
DAVIDE SANGIORGI

Presentata da:
GEJSI VJERDHA

Correlatori:
Chiar.mo Prof.
SAVERIO GIALLORENZO
Dott. GIUSEPPE DE PALMA
Dott. MATTEO TRENTIN

II Sessione
Anno Accademico 2022/23

Minorità è l'incapacità di servirsi della propria intelligenza senza la guida di un altro.

Immanuel Kant, 1784

Sommario

Con il termine *serverless* si indica un nuovo modello architetturale nel campo del cloud computing, caratterizzato dall'esecuzione distribuita, scalabile e basata sugli eventi dei programmi, con la peculiarità che i costi sono proporzionalmente correlati al consumo effettivo delle risorse. Gli sviluppatori scrivono il codice in unità software indipendenti, chiamate *funzioni serverless*, e affidano ai fornitori delle piattaforme la complessa gestione dell'infrastruttura sottostante. Dopo aver esplorato questo paradigma, introduciamo *Fenrir*, un framework che arricchisce il ciclo di sviluppo delle architetture serverless fornendo agli sviluppatori nuovi costrutti di meta-programmazione, detti *annotazioni*, che dotano le funzioni serverless di attributi distinti, e consentono di modellarne il comportamento e le caratteristiche per adattarle alle specifiche esigenze dell'applicazione. *Fenrir* permette di sfruttare tutti i vantaggi del serverless senza sacrificare la compatibilità con progetti già consolidati, poiché è in grado di convertire monoliti esistenti, scritti in *TypeScript*, in architetture serverless.

Abstract

Serverless computing is a new cloud architectural model that promises to execute programs in a distributed, autoscaling, event-driven, and pay-as-you go manner. Developers write the code in self-contained software units, called *serverless functions*, and they let the cloud vendors manage the complex infrastructure underneath. After exploring this architectural model, we introduce *Fenrir*, which is a framework that enriches the development lifecycle of this model by providing developers with new meta-programming constructs, named *annotations*, that imbue serverless functions with distinct attributes, augmenting their behavior and characteristics to align with specific application requirements. *Fenrir* offers developers an avenue to harness the merits of serverless without sacrificing compatibility with established codebases, as *Fenrir* can convert existing *TypeScript* monoliths into serverless architectures.

Contents

1 Introduction

Modern software development requires complex tooling to handle the intricate landscape encompassing the actual code and business logic. Besides the program implementation, there are many components to consider when thinking about all the moving parts needed to make software work in today's dynamic environments, which are supposed to be scalable by design.

Traditionally, software was written in monolithic architectures where the program's components are tightly coupled and tend to create large codebases, as they provide heterogeneous functionalities under the same interface. This paradigm, despite being the oldest, still has some benefits that draw developers' attention especially in the early stages of the development lifecycle. Indeed, these architectures are easier to write because they do not need advanced orchestration mechanisms, they simplify testing and debugging, and they provide the maximum amount of control over the infrastructure: this means that developers have the authority to configure, manage, and optimize each component of the application according to their preferences and requirements. To illustrate, consider an e-commerce application where all features, from user authentication to order processing, are bundled together within a monolithic structure: in this context, the developers decide how all the software units interact and how resources are allocated to ensure optimal performance. Yet, the drawbacks of this paradigm were understood even in the past: a case in point is the *Unix* community, which moved away from it, preferring a modular approach instead [19]. Monoliths are hard to maintain, deploy, adapt to changing requirements and technologies, and they become rigid quickly as reshaping its components may involuntarily cause a chain of problems.

The modular approach, where features are developed in independent software units, gained popularity especially in the cloud services realm. The de facto standard of cloud architectures is represented by *microservices*, which arrange programs into collections of loosely coupled, fine-grained services that communicate through protocols such as HTTP. A microservice is a self-contained service, organized around a specific set of needed business capabilities, that lets developers easily implement a layered architecture using different technologies (e.g., programming languages, databases, etc.). This distributed architectural pattern overcomes the shortcomings of monoliths, as its modular nature makes it scalable, easily deployable, and allows different teams to work together without the risk of taking down the entire structure when a minor issue arrives.

Still, microservices are not the most efficient way of implementing cloud computing architectures, since they are long-running processes whose deployment needs to be carefully planned and managed by configuring the underlying infrastructure. To reduce these operational costs, serverless architectures were recently created.

The serverless paradigm promises to let its users focus simply on the business logic, by shifting to the cloud platform provider the infrastructure management, resources allocation and provisioning, thus enabling software units to scale automatically on demand. In this model, users write the code inside single-purpose units (i.e., programming languages' functions), even smaller than microservices in responsibility, and they set which event will trigger their invocation. This event-driven model of dispatching actions is very efficient, in fact, when a function is triggered, the cloud vendor executes it in an isolated, secure, and short-lived environment. Consequently, users are only billed for the precise resources consumed during this execution period.

Serverless is not free of disadvantages and, while traditional solutions had generations of practitioners and researchers improving the development experience by providing guidelines, best practices, and tools suited for each phase of the development lifecycle (design, programming, debugging, maintenance, etc.), the serverless model is still in its infancy and lacks a similar depth of accumulated wisdom and refinement.

1.1 Objectives

In this thesis, we present *Fenrir*, a framework which aims to enhance serverless programming by simplifying the development of cloud functions through its meta-programming functionalities. Users mark the serverless functions with annotations that signal the *Fenrir* compiler which code transformations are going to be needed and which metadata will be used for their deployment, therefore streamlining different parts of the development lifecycle simultaneously. *Fenrir* transpiles *JavaScript* codebases whose functions are executed through *AWS Lambda*'s *Node.js* runtime. This is accompanied by the seamless management of projects through its user-friendly CLI, ensuring an efficient and graceful path into the adoption of the serverless model.

2 Background

Before introducing the serverless paradigm, we must understand what is cloud computing [10]. Despite its modern connotations, cloud computing isn't a novel concept, in fact, its principles have been established several decades ago: in essence, cloud computing runs different types of workloads within clouds, which are environments that abstract, pool, and share scalable resources across distributed networks. Thus, users are offered on-demand availability of computing power by a third-party provider, without the problems associated with direct active management of IT infrastructures.

2.1 Cloud services

Cloud computing is supplied as a collection of service models, hence, users can arrange different abstraction layers based on their necessities.

1. Infrastructure as a Service (IaaS): these are the most low-level services that can be provided, like bare metal servers, storage, virtual machines, load balancers, etc.

2. Platform as a Service (PaaS): an entire toolkit or development environment made for scaling applications without thinking about the underlying infrastructure. PaaS vendors may offer programming languages execution environments, databases, web servers and many more technologies.

3. Software as a Service (SaaS): this model is in direct contact with the end-user, as it refers to the application software standing on top of the

platforms and infrastructure. Cloud users, such as people using mobile phones, access the software through a subscription fee, without ever needing to install anything because the application is built on top of and balanced through the previously mentioned layers.

Most recently, a new model has appeared, named Function as a Service (FaaS), delivering a cloud service that offers computing runtimes with support for serverless architectures.

Furthermore, the serverless model is also encompassed as Backend as a Service (BaaS), usually accessible via APIs, providing support for many technologies like serverless databases, but these services will not be examined in this thesis.

2.2 Serverless computing

Serverless computing was born due to a very pragmatic reason: workloads in modern applications need to be efficiently managed because they are highly dynamic, meaning that, some software parts need more computing resources than the others and this aspect may vary frequently. In traditional monoliths, or even microservices, developers are concerned with capacity planning, configurations, management, fault tolerance, and scaling of containers, VMs or physical servers.

Serverless architectures abstract way all these concerns by allowing developers to focus solely on writing application logic, and letting the FaaS vendors handle the burden of provisioning and scaling infrastructures.

To achieve this goal, developers write the logic inside software units called cloud functions, which are run in short-lived environments triggered by some kind of event. Cloud functions recall the concept of functions in programming languages, as they associate an input to an output, and as a matter of fact, cloud functions' logic is encapsulated inside the latter. Yet, cloud functions need to be considered as resources rather than instructions of code, as they are software units invoked by the serverless provider and metered on-demand through an event-driven execution model, thus they can be written in different programming languages (e.g., *Go*, *Java*, *JavaScript*) as long as the chosen vendor has adequate support for the language runtime.

When the cloud function is triggered by an event such as HTTP requests, database changes, file uploads, scheduled intervals or various other triggers, the FaaS provider runs the code after initializing an execution environment, which is 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, for example, *AWS Lambda* uses μVMs while *IBM* uses *Docker* containers, nonetheless they all offer lightweight sandboxed containers designed to have fast startup/shutdown times and minimal overhead due to their virtualized nature.

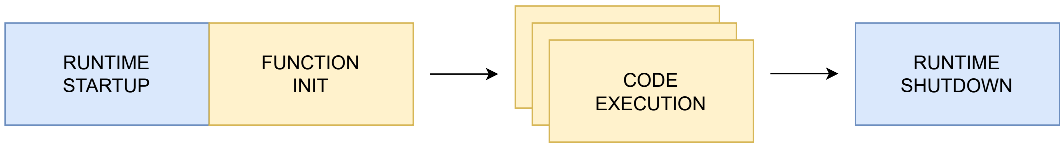

Figure 2.1: Serverless function lifecycle in a warm execution environment (discussed further in Section 2.2.1).

2.2.1 Serverless principles

Serverless is a misnomer, since servers are still in the picture, this computing model must not be confused with other paradigms that do not require an actual server such as peer-to-peer (P2P) [15]. Instead, it is more accurate to interpret it as “less-server”, therefore emphasizing the shift from developers actively managing server infrastructure to entrusting platform providers to manage the underlying server complexities.

Figure 2.1 shows that serverless functions are ephemeral, so they are designed to have a temporary nature, and consequently stateless, so there is no stored information of previous transactions, as they are constructed from scratch each time they are instantiated. These behaviors have several implications that may offer advantages and disadvantages.

Serverless architectures benefits

Reduced costs Serverless at its core gives immediate business value to its users, because it outsources servers management to a vendor, also allowing for economies of scale to take place, as discussed in [6]. Operational expenses decrease significantly, and development costs are optimized since this cloud paradigm offers the most fine-grained billing model in which developers only pay for the actual functions' execution time and not for idle servers.

Elasticity The cloud provider is responsible for autoscaling the capacity on demand, hence resources are handled accordingly to accommodate the required load. For example, consider a web application with two HTTP endpoints, denoted as endpoint A and endpoint B, where A is frequently called while B is very rarely requested. In a traditional monolith, the entire web app would need to manually be scaled to handle unpredictable load spikes of A calls, and even in normal circumstances resources would still be inefficiently allocated just to manage some occasional B requests. By encapsulating their logic inside two different serverless functions, endpoint A can be allocated more resources when it experiences high demand, while endpoint B might cost almost zero due to its occasional requests.

Productivity Developers can focus solely on writing business logic. Exposing units of code only through an event-driven model simplifies back-end development, also alleviating the typical problems related to distributed systems (e.g., multithreading). Moreover, development becomes quicker as deployment is easier and updates are granular, thus increasing the agility of the team and rapid prototyping of new products.

High availability Thanks to the distributed nature of serverless computing, platform providers ensure fault tolerance by redirecting functions across healthy and available zones. Latency is also reduced by deploying functions geographically near the end-users.

Interoperability The serverless paradigm can be used in conjunction with code deployed in traditional styles, such as microservices or monoliths [15]. Therefore, this model can be incrementally adopted by large companies and benefit complex existing systems. Though, start-ups should prioritize following a pure serverless approach because of all the previously mentioned benefits.

Serverless architectures shortcomings

Serverless computing is not flawless, and researchers [5] pointed out some of the limits that are inherent to this paradigm.

State management Considering that there is no server-state, the programming model needs to shift for developers, as numerous applications require storing data for future references. To limit this disadvantage, users may use:

- Temporary data stores or distributed caches, like Redis.
- Databases, both SQL or NoSQL, as long as the connections can be instantiated quickly.
- Vendor-specific strategies, like *AWS*' step functions or *Azure*'s durable functions.

Also, the limited lifetime of serverless functions needs to be contemplated given that they are not designed to support long-running processes, and if misused they can become more costly than traditional solutions.

Cold starts Serverless functions startup times can be a crucial drawback for systems performance, as the overhead for allocating all the resources needed to run a function can cause significant delays. This well-known problem, named "cold start", is particularly emphasized with technologies requiring heavy runtimes, such as *JVM*-based languages. To avoid this constraint, platform providers usually wait some time before dismantling the container, keeping it in a so-called "warm" state, as shown in Figure 2.1, so they are able to run the code without all the initialization overhead: this approach is obviously more costly, but it dramatically improves performances.

Vendor lock-in Any outsourcing strategy is subject to this limitation. Users are bound to the vendors' decisions who wield control over system downtimes, cost changes, loss of functionality, forced API upgrades and many more limits that may, at times, be enforced unilaterally. In addition, each vendor offers different features and workflows, often deeply integrated with their own private services, and as a result switching vendors means updating operational tools (deployment, monitoring, etc.), modifying the code to satisfy the new FaaS interfaces, and most importantly porting chunks of the architecture from one infrastructure to another. To circumvent these issues, programs should be created in a platform-agnostic manner, without relying too heavily on the third-party services offered by the platform provider. Moreover, serverless architectures can be achieved through open-source solutions at the expense of self-hosting parts of the infrastructure.

Complex debugging Monitoring, tracing, logging, and debugging are harder in serverless environments because of their ephemeral and stateless nature. These limits originate from the fact that that cloud environments are difficult to simulate in local contexts, therefore even techniques like integration testing become unpractical. Although entire functions can be timed, there is typically no ability to dig into more detail by attaching profilers, debuggers or APM tools [9].

2.2.2 Serverless use cases

A meta-analysis conducted by Eismann et al. [4] reviewed 89 real-world use cases of serverless architectures and studied their characteristics. Their findings provide insightful values on how this paradigm is being used and how well it performs:

- *AWS Lambda* is the most popular platform provider, taking up 80% of the market share: this is not surprising as they pioneered the most mature and well integrated set of services out of all the cloud vendors.

- *JavaScript* and *Python* are the most used programming languages for cloud functions (each used by 32% of the cases). Most of these architectures depend on a wide variety of cloud services, with the most used ones being cloud storage, cloud database, API gateway and cloud pub-sub.

- Cost savings seem to be the strongest motivator for adopting serverless computing, still, other driving factors are reduced operation effort, the scalability and performance gains.

- The overall trend of serverless architectures is to feature unpredictable on-demand workloads, typically triggered through lightweight (<1MB) HTTP requests.

The first two points also serve to understand how *Fenrir* operates: we chose to compile *JavaScript* codebases written following *AWS Lambda*'s interfaces due to their ever-growing popularity. Furthermore, the meta-programming offered through *Fenrir*'s annotations can be expanded to support different platform providers.

To simplify the development of serverless architectures even more, *Fenrir* not only manipulates source codes, but it also generates metadata used by the *Serverless Framework* [16] (referred to as *SLS* in this thesis): this framework streamlines the operational efforts by providing a simple abstraction layer that can deploy with ease to all major cloud providers by tweaking a few configuration details. Thus, *SLS* helps in breaking the shortcomings of vendors' control, and it also brightens the developer experience by extending workflows through CLI tools, debugging facilities and useful plugins.

2.3 Server-side JavaScript

“Any application that can be written in JavaScript, will eventually be written in JavaScript.”

As time passes, this quote by Jeff Atwood becomes more and more accurate: in fact, it should be expanded in scope to include all types of software, as *JavaScript* continues to evolve and keeps being one of the most popular programming languages of all time.

Despite its humble beginnings, its versatility and simplicity have taken the language a long way, giving rise to a vibrant open-source ecosystem that has enlarged its domain beyond web browsers: *JavaScript* is now widely used in diverse fields including game development, IoT, data analysis, and numerous other contexts.

Probably, the most important contribution to the language has been the advent of *Node.js* in 2009 [12], which is a runtime environment that executes *JavaScript* outside a web browser. This technology played a pivotal role in reshaping users’ perceptions of *JavaScript*: it transitioned from being regarded as a mere scripting language to being recognized as a capable general-purpose one.

Node.js stands apart from traditional programming language runtimes, and, instead, inherits the strengths of *JavaScript* and builds upon them, making this environment unique in many ways. *Node.js* operates on a single-thread event loop, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context switching.

This event-driven design optimizes throughput and scalability in architectures bound to many asynchronous I/O operations, but it is limited in CPU-heavy environments given the single-threaded nature of *Node.js*. For this reason, *Node.js* shines in serverless architectures more than traditional ones, since it is a great runtime to support the ephemeral and stateless cloud functions, where users do not need to think about the orchestration of distributed systems.

2.3.1 TypeScript

TypeScript is a superset of *JavaScript* with an additional fundamental layer: the type system.

Just like *Node.js*, *TypeScript* is a pretty unique technology. Indeed, this language was born to fix many of the quirks present in *JavaScript*, which, given its popularity, started being used in codebases with hundreds of thousands of lines of code, even though, it was never intended for such use cases, hence, its error-prone nature started appearing at runtime with all its oddities and surprises. While other tools tried replacing it, and consequently failed (e.g., *CoffeeScript*), *TypeScript* embraced it and added a compile-time type checker to drastically lower the chance of bugs.

TypeScript is a strongly typed programming language, but it is also designed to support existing *JavaScript* codebases, and as a result many of the strict type checks performed can be loosened up to compile regular *JavaScript*. Furthermore, *TypeScript* guarantees to preserve the runtime behavior, even if the compiler raises type errors, allowing the transition between the two languages to happen without subtle differences. Once the compiler has finished checking the code, the types are erased and the resulting code has no type information.

Normally, *TypeScript* is very strict during its checks, so other than blocking the weird *JavaScript* parts, it also provides guards for writing code at enterprise level, ensuring that everything works even in big projects with multiple teams, as long as the established type definitions are followed.

```typescript
/* Guards against JavaScript's quirks */
if ('' == 0) {
}
// Error: This comparison appears to be unintentional because the
// types 'string' and 'number' have no overlap.

console.log(4 / [])
// While this is a syntactically-legal instruction that logs `Infinity`,
// TypeScript will issue an error because it is a nonsensical operation.

```

```
/* Strict type checks */
type User = {
  firstName: string
  lastName: string
  role: 'Professor' | 'Student'
}

const user: User = {
  firstName: 'Angela',
  lastName: 'Davis',
  role: 'Professor',
  name: 'Angela Davis',
}
// Error: Object literal may only specify known properties
// and 'name' does not exist in type 'User'.
```

Listing 2.1: *TypeScript*'s type checks

Listing 2.1 previews a very important principle of *TypeScript*, its structural type system. Although, most strongly typed languages implement a nominal type system, where two types are considered equal when their names correspond, *TypeScript* had to be more pervasive to enhance *JavaScript*'s dynamic duck typing [3], and as a consequence it implements structural checks, where, rather than the name, the shapes of the values are compared.

```
type Dollar = number
type Euro = number
function foo(n: Dollar)
let e: Euro
// This function call raises an error in a nominal system
// because `foo` only accepts an argument of type `Dollar`,
// but works completely fine in a structural system,
// since the two types share the same shape of `number`.
foo(e)
```

To be more precise, *TypeScript*'s type system is not entirely structural, as it offers nominal typing-like mechanisms to simplify the writing of new definitions, by making types unique symbols across their contexts, which is especially useful for creating recursive data structures.

All these key factors (i.e., strictness, compatibility with *JavaScript*, structural typing), have greatly boosted *TypeScript*'s popularity at every scale of software development. On top of that, it introduces powerful features that improve developers' experience:

- Types inference support vastly superior in comparison to most of the commercial general-purpose programming languages.
- Enriched build pipeline. Before *TypeScript* existed, projects were developed through complex combinations of different tools, especially if the developers preferred to use newer features of *JavaScript* that were not supported by the runtime yet. *TypeScript* simplifies these workflows since it provides an inclusive build system which compiles even the experimental features of the language to older versions without the need for polyfills or other technologies.
- Better tooling through open-source contributions. For example, *TypeScript* was the reason behind the birth of the *Language Server Protocol* (*LSP*), which enriches code editors with language intelligence tools such as code completion, syntax highlighting, error marking, refactoring routines and many more features. Traditionally, all this work had to be repeated for each programming language as they all built upon different APIs, but now all major programming languages follow this specification, which decouples the language services from the editors. This is just one of the tools built because of and for *TypeScript*, but it should not be underestimated since it improves developers' experience significantly.

In serverless computing, where functions need to be self-contained, concise, and easily deployable, *TypeScript* strengths are particularly valuable, since it encourages developers to write robust and maintainable code. This is convenient since cloud functions are harder to test and debug, and having compile-time errors reduces maintenance costs and saves a lot of time.

Fenrir uses the parser offered by the *TypeScript* API to handle AST traversals and manipulations, so it parses both *TypeScript* and *JavaScript* codebases.

3 Implementation

This chapter presents an in-depth description of how *Fenrir* operates by reviewing each step of its compilation process and analyzing the concept of annotations which represent the foundation of this framework. Moreover, it presents practical use cases to demonstrate how much it can simplify developers' experience by allowing them to focus on the core logic of their applications while effortlessly benefiting from the serverless paradigm.

3.1 Overview

Fenrir is a transpiler which enhances serverless programming by introducing the concept of annotations. Annotations are an abstraction layer that the developers can unobtrusively use to apply code transformations and metadata generation to a given application, which will be deployed to a serverless platform.

To achieve this goal, we used the *TypeScript* [18] compiler API which lets us manipulate sources with ease, and *SLS* [16] which uses the generated metadata to deploy to *AWS Lambda*.

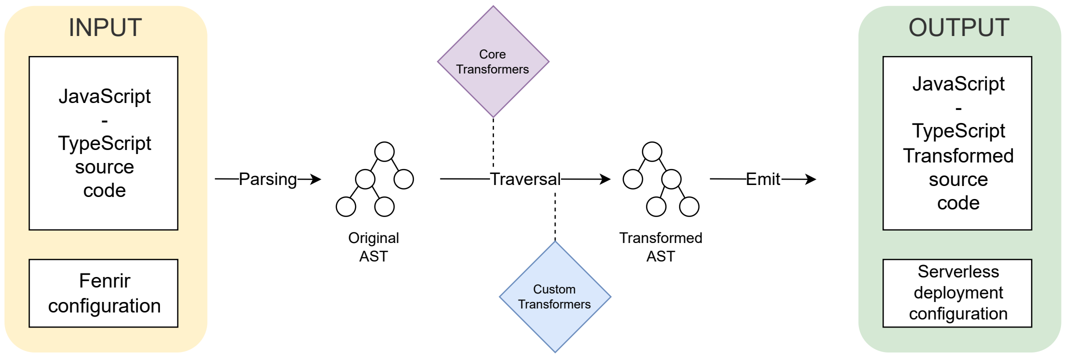
INPUT
JavaScript - TypeScript source code
Fenrir configuration
Parsing
Original AST
Core Transformers
Traversal
Custom Transformers
Transformed AST
Emit
OUTPUT
JavaScript - TypeScript Transformed source code
Serverless deployment configuration

Figure 3.1: Transpiler pipeline.

The transpilation pipeline, depicted in Figure 3.1, starts with the parsing of the input source code, 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, the output code is saved and the related metadata is also appended to a `serverless.yml` file which specifies function deployment properties (e.g., the address to invoke a given function).

It is important to notice that *Fenrir* does not lock developers in managing their functions only through its tools, instead, its primary objective is to facilitate the incremental adoption of the serverless paradigm.

3.1.1 Brief example: Monolith to Serverless conversion

In Listing 3.1, 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 `$HttpApi` annotation. The `generateReport` function is instead a backend one, which we want to run at pre-established intervals: to obtain this behavior, we use the `$Scheduled` annotation to specify that it shall be run every two hours. All these annotations are explored in detail in Section 3.3. Using *Fenrir*, we translate the code of Listing 3.1 into the serverless codebase of Listings 3.2– 3.3.

```
/**
 * $Fixed
 * $HttpApi(method: "GET", path: "/orders/report")
 */
export async function processOrder(orderId) {
  // ... processing logic ...
  console.log(`Processing order ${orderId}`)
  return order
}
/** $Scheduled(rate: "2 hours") */
export async function generateReport() {
  // get the processed data and generate report
  console.log("Generating report")
}
```

Listing 3.1: Source Code.

```
export async function processOrder(event) {
  const orderId = event.orderId
  // ... processing logic ...
  console.log(`Processing order ${orderId}`)
  return {
    statusCode: 200,
    body: JSON.stringify(order),
  }
}
// The implementation of `generateReport`
// is omitted as it remains unchanged.
```

Listing 3.2: Generated Code.

```yaml
processOrder:
  handler: output.processOrder
  events:
    - httpApi:
      method: GET
      path: /orders/report
generateReport:
  handler: output.generateReport
  events:
    - schedule:
        rate: 2 hours
```

Listing 3.3: Generated deployment configuration.

The generated code includes both the `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 3.2, we create a JSON object with a status code and a body that contains a serialized version of the value held by the variable `order`, which holds the value returned by the function in the source codebase. The other notable element in the YAML code found in Listing 3.3, 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.

This brief example introduces features that will be thoroughly described, but it does not contain any real logic. A non-trivial example, named *Hati*[1], is present in the *Fenrir* public repository, yet, it is omitted from this thesis due to its size.

Hati converts a monolithic email message analysis program, into an event-driven serverless pipeline where each function has the same core logic as the monolithic counterpart, but they are manipulated to have all the benefits offered by the serverless paradigm.

[1]In Norse mythology, Fenrir has two sons, Hati and Sköll.

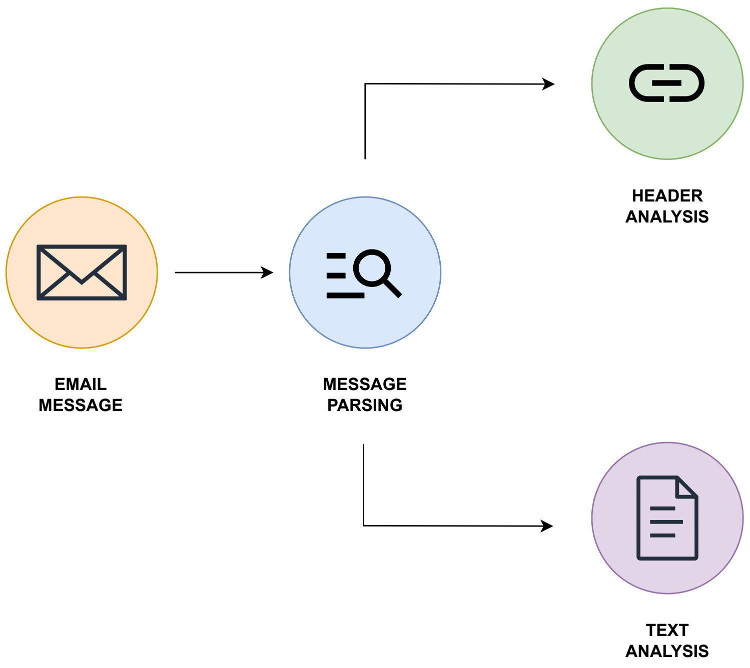

Figure 3.2: *Hati*'s pipeline.

3.2 Parsing

Fenrir requires a configuration file that must be named `fenrir.config.json` to understand where to operate:

```json
{
  "files": ["input/source.ts"],
  "serverlessConfigPath": "input/serverless.yml",
  "outputDirectory": "output",
  "annotations": {
    "CustomAnnotation": "annotations/custom-annotation.ts"
  }
}
```

Listing 3.4: *Fenrir* configuration

The `files` field accepts an array of filenames or a directory, and it is used to represent which files will be parsed by the transpiler.

The `serverlessConfigPath` field points at the input *SLS* configuration that must contain some mandatory input metadata, such as the region which the functions will be deployed on.

The `outputDirectory` field indicates where the emitted files will be placed.

The `annotations` field locates custom annotations as it takes an object where the keys represent the new names and the values refer to their associated implementations.

Fenrir's CLI After creating the configuration, *Fenrir* can be started through its CLI tool by optionally passing as a flag the directory in which it is contained.

```
# defaults to the current directory (.) for its file lookup...
> fenrir
# ...or uses a custom path
> fenrir -g input-directory
```

Furthermore, the CLI offers the `init` sub-command to ease the setup needed for the entire pipeline by generating the necessary boilerplate and configuration files.

AST Traversal

Through the *TypeScript* compiler API, each sources' AST is traversed by a visitor function which collects some data (e.g., dependencies imports) and examines certain types of node in order to process annotations, namely *exported function declarations*. Lookups are restricted only to this syntactic category for two reasons:

- `export` ensures the functions are public and ready to be deployed.

- `function declarations` minimize the code needed to control the AST. Considering that *JavaScript* provides three distinct ways of declaring a function, accommodating all of these variations would inevitably result in a threefold expansion of the manipulation code. Moreover, function declarations are the idiomatic technique to write top-level functions.

```
// Skipped
const a = 2;
// Skipped
for (const b of [1, 2, 3]) {}
// Processed
export function foo() {}
// Skipped
export const fiz = () => {}
// Skipped
export const bar = function() {}
```

Listing 3.5: Examples of processed or skipped nodes.

The processed functions have their annotations examined, and their bodies are visited to handle AST transformations.

3.3 Annotations

Annotations are syntactical units, or keywords, enclosed within JSDoc comments, each associated with their respective transformer. They can be expressed using a BNF-like syntax:

$$\begin{aligned}
\text{Annotation} &::= \texttt{"\$"}\langle\text{Name}\rangle[\texttt{"("}\langle\text{Parameters}\rangle\texttt{")"}] \\
\text{Name} &::= [\text{a-zA-Z0-9\_}]+ \\
\text{Parameters} &::= \langle\text{TypeScriptObject}\rangle \\
\text{TypeScriptObject} &::= \ldots
\end{aligned}$$

This representation does not intend to provide a formal and complete definition of the syntax of annotations. Nevertheless, it serves to offer a general intuition of how they can be written within the code.

```
/** $Foo */

/** $Bar(param: "first") */

/** $Fiz(first: true, second: [1, 2, 3], third: (a, b) => a + b) */

/**
 * Annotations enrich docs...
 * $Foo
 * ...and can even be multiline.
 * $Bar(
 *    param: "fiz"
 * )
 *
 * all these explanatory texts are not
 * harmful, as they are simply ignored.
 */
```

Listing 3.6: Generic examples of annotation.

Annotations are designed to avoid cluttering *JavaScript* code with new syntax, as they can only be written inside JSDoc comments. Thus, they serve a dual-purpose, they provide the functionality needed by *Fenrir*, and they also serve as supplementary documentation for the codebase.

Listing 3.6 provides a valuable insight on how annotations may be parameterized to modify transformers' functionality. In fact, arguments undergo parsing as a *TypeScript* object literal, hence, they are as powerful as regular objects [11] meaning that even complex structures (arrays, functions, etc...) can be used during the compilation step to enrich transformations.

Another crucial feature of annotations is their composability, as multiple annotations can be written within the same JSDoc, essentially defining dedicated compilation pipelines by passing the output of a transformation as input for the subsequent ones: to facilitate this process, each source file may be visited multiple times.

3.3.1 Core Annotations

Fenrir offers four core annotations whose transformers can handle code manipulation, deployment metadata or a combination of both functionalities: by pipelining annotations, we can effectively utilize the strengths of different approaches.

Table 3.1: Core annotations

Name	**Code**	**Metadata**
$Fixed	Yes	Yes
$TrackMetrics	Yes	No
$HttpApi	No	Yes
$Scheduled	No	Yes

$Fixed

`$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. To achieve this conversion, code is handled as follows:

- The monolithic functions' parameters are mapped to a single `event` parameter in order to adhere to *AWS Lambda* serverless functions' signature.
- The monolithic functions' return statements change to match the shape of the response expected by the platform, by creating an object with a status code (`200`) and a body that contains a serialized version of the initially returned value.
- Early return statements and throw statements are modified similarly, but the status code represents a client error (`400`).

`$Fixed` has no mandatory parameters, however, all the specified arguments will be passed as metadata for the function deployment.

```
/** $Fixed(timeout: 10) */
export async function foo(id) {
  if (!isValid(id)) {
    throw new Error('Something went wrong')
  }

  const data = await query()

  return data
}
```

Listing 3.7: Input code for `$Fixed`

```
/** $Fixed(timeout: 10) */
export async function foo(event) {
  const id = event.id

  if (!isValid(id)) {
    return {
      statusCode: 400,
      body: JSON.stringify({
        error: "'Something went wrong'",
      }),
    }
  }

  const data = await query()

  return {
    statusCode: 200,
    body: JSON.stringify(data),
  }
}
```

Listing 3.8: Output code from `$Fixed`

```
functions:
  foo:
    handler: output/source.foo
  timeout: 10 # default is 6 seconds
```

Listing 3.9: Generated Metadata through `$Fixed`

$TrackMetrics

`$TrackMetrics(namespace: string, metricName: string, metricValue?: ts.Expression)` generates code that monitors and logs the functions' resource usage by also importing the necessary dependencies, i.e., for *AWS Lambda* it uses and injects the *CloudWatch* dependency. These are the required properties:

- The function declaration must be `async`. Preceding this annotation with `$Fixed` makes it automatically `async`.
- `namespace` and `metricName` are mandatory strings. The former represents the namespace instantiated on the cloud through *AWS*.
- The third parameter, if present, must be the same as one of the variables' identifiers.

We complete the explanation with an example:

```
import { query } from './local'

/**
 * $TrackMetrics(namespace: 'shop', metricName: 'sell', metricValue: size)
 */
export async function processOrder(id) {
  const order = await query(id)
  const size = order.size
  // ...more logic...
  return size
}
```

Listing 3.10: Input code for `$TrackMetrics`

```
import { query } from './local'
import { CloudWatch } from 'aws-sdk'

/**
 * $TrackMetrics(namespace: 'shop', metricName: 'sell', metricValue: size)
 */
export async function processOrder(id) {
  const order = await query(id)
  const size = order.size
  await new CloudWatch()
    .putMetricData({
      Namespace: 'shop',
      MetricData: [
        {
          MetricName: 'sell',
          Timestamp: new Date(),
          Value: size,
        },
      ],
    })
    .promise()
  // ...more logic...
  return size
}
```

Listing 3.11: Output code from `$TrackMetrics`

Listing 3.11 shows the generated boilerplate to enable logs inside the function, and an additional import statement is included at the top of the file to address its previous absence. `$TrackMetrics` puts the code in a context-aware manner, placing it after the variable declared as `metricValue`.

Supposing a typographical error was written instead of `size`, the following error message would appear:

```
'$TrackMetrics' can only receive an identifier as
a value for the 'metricValue' parameter like
'id' | 'order' | 'size'
in function 'processOrder' defined here:
--> input/source.ts:6
```

$HttpApi

`$HttpApi(method: string, path: string, ...)` generates the metadata needed to make the function available as an HTTP endpoint.

- The `method` parameter represents the desired HTTP method (`GET`, `POST`, etc...).

- The `path` parameter represents the URL path where the function will be available.
- The optional parameters are meant to customize the behavior of the HTTP endpoint further, for example, by setting up Cross-Origin Resource Sharing (CORS) headers.

```
/**
 * $HttpApi(method: "POST", path:
     "/users/create")
 * $HttpApi(method: "PUT", path:
     "/users/update")
 */
export function foo() {}
```

```
foo:
  handler: output/source.foo
  events:
    - httpApi:
        method: POST
        path: /users/create
    - httpApi:
        method: PUT
        path: /users/update
```

$Scheduled

`$Scheduled(rate: string, ...)` generates the metadata needed to make the function run at specific dates or periodic intervals.

- The `rate` parameter is a rate or cron expression which schedules when the function should be triggered.
- The optional parameters are meant to customize the behavior of the scheduled event further, for example, by specifying multiple schedule expressions and giving it a description.

```
/**
 * $Scheduled(rate: "rate(2
     hours)")
 */
export function foo() {}
```

```
foo:
  handler: output/source.foo
  events:
    - schedule:
        rate: rate(2 hours)
```

3.3.2 Custom Annotations

Fenrir is not bound to its core annotations, as it endorses the creation of new ones to fit custom requirements and usages.

In order to inform *Fenrir* of the new annotation name and its transformer implementation, the configuration file (i.e., `fernrir.config.json`) must be updated:

```json
{
  ...
  "annotations": {
    "IoT": "annotations/iot-impl.ts"
  }
}
```

Fenrir encourages strict type-safety by offering a type definition for custom transformers:

```typescript
// in 'annotations/iot-impl.ts'
import type { CustomTransformer } from 'fenrir-core'

type IotTransfomer = CustomTransformer<'IoT', { sql: string }>

const transformer: IotTransfomer = (
  node,
  context,
  annotation
) => {
  // ...implementation...
}

// custom transformers must be exported as `default`
export default transformer
```

Listing 3.12: Custom transformer for a new `$IoT` annotation.

Analyzing custom transformers' signature also provides insights on how core annotations are implemented, since their definitions are almost identical.

The `node` argument represents the function declaration in the AST which was marked with the annotation. It contains all the relevant information associated with this type of AST node, including the function's body, name, parameters, and other relevant details.

The `context` argument represents the *TypeScript* transformation context, which is a very powerful object containing:

- Methods for source code manipulations, such as `context.factory.updateIfStatement()`.
- The typechecker, which is an essential module integrated in *TypeScript*, useful for working with symbols: it has access to every type of each node and details about the project dependencies.
- Miscellaneous methods to handle lexical environments and compiler options.
- *Fenrir* utilities that facilitate the development of transformers and of metadata handling.

The `annotation` argument is an object containing the annotation name and a record of all the arguments passed to it. Its type definition is inferred from the previously established `CustomTransformer` schema.

A transformer's return type can be omitted as it is inferred, however, if it is manifested to provide a stricter signature, it is restricted to this subset:

```
type TransformerReturnType =
  | ts.SourceFile
  | ts.FunctionDeclaration
  | undefined
  | void
```

Hence, transformers may perform three types of tasks: modifying the entire source file, which is useful for updating import declarations or other AST nodes, altering the function itself, or solely modifying the metadata without changing the source code.

3.4 Emit

The last step of *Fenrir*'s pipeline involves emitting the transformed files and metadata. Files are ejected in a single output folder, with the default name being *functions*: it is important to give it an insightful name, since, users are encouraged to review and potentially modify the emitted code if they aren't satisfied with the transformations.

The transformed code may not look as the input code, as it is formatted through an opinionated set of conventions which imposes properties like the usage of semicolons or the number of spaces: in order to make it adhere to personal formatting preferences, users should use popular tools such as *ESLint* and *Prettier*.

Generated metadata is appended to a `serverless.yml` placed at the project root, and through the *SLS* CLI it may be deployed to the serverless platform.

3.4.1 Error handling

Fenrir tries to provide as many informations as it can when it meets errors.

Configuration mistakes cause the program to panic, for example:

```
Since you are providing a list of files, a `serverless.yml` must
be provided to the transpiler to generate the needed metadata.
```

Annotations syntax errors provide another set of logs:

```
$Schdle
 ^^^^^^
 Unknown annotation name

$Scheduled(
          ^
          Invalid bracket

$Fixed(foo: )
      ^^^^^^^
      Invalid syntax for annotation parameters
```

Transformers can also emit specific errors only related to their domain, such as:

```
'$HttpApi' must receive both 'method' and 'path' as parameters
in function 'foo' defined here:
--> input/source.ts:6
```

4 Conclusions

After a brief introduction, we discussed all the necessary knowledge needed to understand this thesis in Chapter 2, and finally we presented how *Fenrir* works, by thoroughly examining its inner components in Chapter 3.

The most important concept to recall about this framework is its goal, and how it tries to achieve it. *Fenrir* wants to enhance the serverless programming model through its meta-programming features. In fact, its core annotations operate very differently from each other, as they are related to distinct domains, but they still can be combined to enrich the development experience. Thus, *Fenrir*'s users are not only inheriting all the benefits offered by serverless architectures, but they are also accessing all its powerful code transformations that can save precious development time.

Fenrir was also designed to help migrate existing monolithic codebases into serverless ones through an incremental approach, meaning that users do not need to convert their entire project, but they can partially adopt this new computing paradigm by iteratively annotating which components should be deployed as independent functions. Additionally, it is crucial to understand that the code emitted by *Fenrir* is not set in stone, instead, it can be actively modified and even used as a starting point of a serverless codebase. *Fenrir* should be viewed as a coding assistant rather than a fully comprehensive tool to write serverless functions.

Fenrir's most notable limitation is the lack of more core annotations, which stems from the substantial time investment required to establish the framework's foundations, reflect on valuable transformers and write their implementation. Still, users are encouraged to create their custom annotations tailored to address their specific use cases, which can include scenarios that cannot be foreseen by *Fenrir* due to their unique nature.

Another important decision to discuss is our rationale behind selecting *TypeScript*. Many other programming languages offer meta-programming compile-time capabilities, like *Rust* macros, *C++* templates or *Zig* comptime. Such systems are far superior to our annotations, yet, these low-level languages are not suitable for the high-level programming that the serverless functions are supposed to entail, in fact, most of these languages do not have the runtime support from any major platform provider. Moreover, even high-level languages with similar compile-time features, like *Scala*, demand to be monitored with caution, because of their heavy runtime (i.e., *JVM* for *Scala*) initialization spin-ups that may cause expensive cold starts.

TypeScript, enriched with our annotations, perfectly fits the serverless model as it runs, after its strict static type checks, on *Node.js*, which is a well-suited lightweight environment designed to handle distributed services, as discussed in Section 2.3.

4.1 Comparison with previous studies

One of the features offered by *Fenrir* is the fact that it can port an existing monolith to a serverless platform as shown in Section 3.1.1. This process, named "FaaSification", has captured the attention of researchers who have explored and presented various solutions to this challenge.

The first attempts were developed for *Python* and *Java* monoliths, with the tools being named respectively *Lambada* [17] and *Termite* [2]. They are fairly limited, as their emitted code cannot correctly process inputs, and they do not support the use of global variables within the serverless functions.

Instead, let us focus on the technologies built around *JavaScript*:

Node2Faas [1] It was the first *JavaScript* "FaaSifier". It converts methods of the *Node.js* monolith into serverless functions and replaces their bodies with an API call to the target FaaS system. This tool has many weaknesses: most notably, it cannot resolve either code or package dependencies, which is a considerable constraint since many cloud functions work by using third-party services.

DAF AND **M2FaaS** [14, 13] They were built by the same team of researchers, the latter, *M2FaaS*, represents an improved iteration of the former. They introduce a concept similar to our annotations, through which users can mark arbitrary parts of their code with the dependencies needed to be deployed. The outcome is a serverless monolithic application hybrid. However, due to technical limitations, the development effort to write their marking constructs is remarkable, as users are compelled to write configuration details inside their business logic.

FaaSFusion [8] This is the closest work to *Fenrir*. *FaaSFusion*'s users also mark their functions with annotations and some associated code transformations occur. This framework brings infrastructure-as-code concepts into the function source, but this approach has evident shortcomings since it heavily relies on heuristics for its foundations. For example, it offers an annotation named `@Warmup` which injects an algorithm to avoid cold starts by periodically pinging the function through a *CloudWatch* event: however, platform providers have suggested avoiding these so-called "warmers", because they are mostly ineffective, break serverless principles, and can increase costs very quickly.

The main obstacle that these tools do not overcome is that they are not designed to handle codebases beyond the trivial examples which they present to support their cases. In comparison, *Fenrir* is very powerful, and its scope is broader.

4.2 Future directions

We believe *Fenrir* could be improved further by expanding its features or by building tools around it. We close this thesis by listing a few of the enhancements *Fenrir* could receive.

4.2.1 More annotations

Fenrir should have more core annotations. Also, while the existing annotations currently do not exhibit conflicts with each other, it is prudent to anticipate potential conflicts that could arise when new annotations are introduced. Consequently, incorporating new constraints becomes essential to prevent such conflicts and ensure the smooth coexistence of annotations within the framework.

Fenrir should add support for marking not just top-level functions, but also other AST nodes. This feature would enrich the framework further, as users would have more freedom to change the emitted code. These new annotations may be inside the function itself, marking nodes inline, or even at the source level. Additionally, they may be related to a particular annotation, thus, creating sets of domain-specific annotations. For example, `$Fixed`, whose scope is related to "FaaSification", could have *children* annotations that would issue a warning if used under other top-level annotations.

```
/** $Fixed */
export async function foo() {
  const data = await query()
  // new inner annotation, which changes parts of the emitted code
  /** $StatusCode(code: 201) */
  return data
}
```

4.2.2 Formalization

Many parts of this framework should be formalized, similarly to the work conducted by Kallas et al. [7], as this would give solid foundations to *Fenrir*:

- Annotation syntax and semantics, including defining the structure of annotations, allowed parameters, and their meanings, because we only presented a preliminary insight in Section 3.3.
- Composition semantics to spot potential conflicts while using multiple annotations.
- Transformations algorithms to ensure correctness, predictability, and consistency in the compilation process, given some notion of behavioral correspondence.

4.2.3 Linter

Currently, there is no standalone linter or set of rules for existing ones that captures serverless principles and enforces them on projects. A new linter may analyze the code, flag architectural smells, and warn users whenever they are breaking best practices. For example, the linter may issue an error if there is a serverless function invoking another one, as this could lead to increased costs, more debugging complexity, and a breach of the isolation principle.

Furthermore, a new linter could be built specifically for *Fenrir*, flagging errors and warnings when a developer misuses annotations. Linting may take place either before the transpilation process or afterward, targeting the emitted directory.

Bibliography

1. L. Rebouças de Carvalho and A. e. Favacho de Araújo. "Node2FaaS: Automatic NodeJS Application Converter for Function as a Service". In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. CLOSER 2019. SCITEPRESS - Science and Technology Publications, Lda, Heraklion, Crete, Greece, 2023, pp. 271–278. ISBN: 9789897583650. DOI: `10.5220/0007677902710278`. URL: `https://doi.org/10.5220/0007677902710278`.
2. S. Dorodko and J. Spillner. "Selective Java code transformation into AWS Lambda functions". In: *ESSCA@UCC*. 2018. URL: `https://api.semanticscholar.org/CorpusID:84836602`.
3. *Duck typing*. Wikipedia. URL: `https://en.wikipedia.org/wiki/Duck_typing`.
4. S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. "A Review of Serverless Use Cases and their Characteristics". *CoRR* abs/2008.11110, 2020. arXiv: `2008.11110`. URL: `https://arxiv.org/abs/2008.11110`.
5. J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. "Serverless Computing: One Step Forward, Two Steps Back". *CoRR* abs/1812.03651, 2018. arXiv: `1812.03651`. URL: `http://arxiv.org/abs/1812.03651`.
6. E. Jonas, J. Schleier-Smith, V. Sreekanti, C. -che Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". *CoRR* abs/1902.03383, 2019. arXiv: `1902.03383`. URL: `http://arxiv.org/abs/1902.03383`.

7. K. Kallas, H. Zhang, R. Alur, S. Angel, and V. Liu. "Executing Microservice Applications on Serverless, Correctly". *Proc. ACM Program. Lang.* 7:POPL, 2023. DOI: 10.1145/3571206. URL: https://doi.org/10.1145/3571206.
8. R. Klingler, N. Trifunovic, and J. Spillner. "Beyond @CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns". In: *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. WoSC '21. Association for Computing Machinery, Virtual Event, Canada, 2021, pp. 23–28. ISBN: 9781450391726. DOI: 10.1145/3493651.3493669. URL: https://doi.org/10.1145/3493651.3493669.
9. P. Leitner, E. Wittern, J. Spillner, and W. Hummer. "A mixed-method empirical study of Function-as-a-Service software development in industrial practice", 2018. DOI: 10.7287/peerj.preprints.27005.
10. P. M. Mell and T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011.
11. Mozilla Developer Network. *MDN Web Docs - Objects*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object.
12. *Node.js*. OpenJS Foundation. URL: https://nodejs.org.
13. S. Pedratscher, S. Ristov, and T. Fahringer. "M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks". *Future Generation Computer Systems* 135, 2022, pp. 57–71. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.04.021. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22001509.
14. S. Ristov, S. Pedratscher, J. Wallnoefer, and T. Fahringer. "DAF: Dependency-Aware FaaSifier for Node.js Monolithic Applications". *IEEE Software* 38:1, 2021, pp. 48–53. DOI: 10.1109/MS.2020.3018334.
15. *Serverless computing*. Wikipedia. URL: https://en.wikipedia.org/wiki/Serverless_computing.
16. Serverless Framework. *Serverless Framework: Build applications on AWS Lambda, Azure Functions, Google Cloud Functions, and more*. URL: https://www.serverless.com/.

17. J. Spillner. "Transformation of Python Applications into Function-as-a-Service Deployments". *CoRR* abs/1705.08169, 2017. arXiv: `1705.08169`. URL: `http://arxiv.org/abs/1705.08169`.
18. TypeScript. *TypeScript: JavaScript that scales.* URL: `https://www.typescriptlang.org/`.
19. *Unix philosophy.* Wikipedia. URL: `https://en.wikipedia.org/wiki/Unix_philosophy`.

Ringraziamenti

Voglio ringraziare prima di tutto i miei due relatori che mi hanno permesso di presentare questo mio umile progetto e mi hanno lasciato piena libertà decisionale su ogni aspetto. Le loro indicazioni sono state fondamentali: ringrazio il prof. Davide Sangiorgi che con i suoi consigli mi ha fatto da mentore in questi anni, e ringrazio il prof. Saverio Giallorenzo che, da grande stachanovista quale è, mi ha seguito e guidato attivamente durante tutto lo sviluppo. Li ringrazio soprattutto perché sono tra le persone più competenti che io abbia mai incontrato e perché mi hanno introdotto a quei campi di questa scienza che mai avrei pensato potessero suscitare tanto interesse in me. Inoltre, ringrazio anche il Dr. Matteo Trentin ed il Dr. Giuseppe De Palma, che oltre ad essere due dottorandi brillanti e pieni di idee nuove, mi hanno sempre dato una mano.

Ringrazio la mia famiglia per non avermi messo nessun tipo di pressione, e per avermi accolto sempre a casa con gioia. Mi hanno reso un uomo maturo ed indipendente.

Ciò che mi rimarrà di questi tre anni a Bologna, più che le nozioni, saranno le persone e la valanga di ricordi che ho creato con i miei amici. Ho conosciuto persone incredibili, nel bene o nel male, che solo Bologna poteva offrire. Il gruppo che si è venuto a formare era pieno di personaggi assurdi a cui auguro il meglio. Ringrazio Tommaso, Gabriele, Simone, Ilaria, Greta, Arianna, Francesco e tutti gli altri che non riesco a citare perché sono troppi.

Infine, ringrazio i miei migliori amici Luca, Giovanni e Giulio, con cui ho avuto la fortuna di condividere la casa per tre anni. Sarà impossibile trovare

dei coinquilini migliori, e sarà impossibile scordarsi di tutte le risate che ci siamo fatti.