SFIOC: a Platform to Support Service Dependency Injection in Serverless Functions

SFIOC: a Platform to Support Service Dependency Injection in Serverless Functions

Andrea Sabbioni, Luca Foschini
Department of Computer Science and Engineering, *University of Bologna*, Bologna, Italy
Email: { andrea.sabbioni5, luca.foschini}@unibo.it

***Abstract*—Function as a Service (FaaS) is a serverless cloud computing model that enables customers to encapsulate their business logic in functions. The platform automatically executes these functions each time a specified event occurs and are terminated once the triggering event is processed. Function as a Service (FaaS) workflows are often supported by ready-to-use and fully managed services hosted by cloud providers belonging to the so-called Backend as a Service (BaaS). The integration of these two serverless models offers comprehensive support to developers, enabling them to focus on business logic development while benefiting from a fully managed and automated infrastructure. However, state-of-the-art FaaS platforms currently lack direct support for integrating Backend as a Service (BaaS) services in FaaS functions, often resulting in service calls being hard-coded within function code. This approach reduces the modularity of functions, enforces vendor lock-in, and negatively impacts the performance of FaaS workloads.**

In this work, we propose SFIOC, an architecture that facilitates the integration of BaaS services into FaaS functions through Dependency Injection. SFIOC enhances existing FaaS solutions by enabling dynamic and at-runtime resolution of function service dependencies. SFIOC automates dependency management, thereby improving the maintainability and modularity of serverless functions while preserving workflow performance.

The capabilities of our solution are demonstrated through an extensive testbed that encompasses the enhancement with SFIOC of a public cloud provider and an open-source-based private edge environment.

***Index Terms*—Serverless, Dependency Injection, FaaS, BaaS, Cloud Continuum, Cloud Edge**

I. Introduction

The increasing complexity of modern deployments and IT infrastructure deems solutions that alleviate burdens associated with infrastructure management, enabling customers to focus on business logic development. These abstractions become even more relevant in cloud continuum hosting environments, characterized by the concurrent adoption of multiple platforms, both privately hosted and offered by different vendors, with heterogeneous performance and technology availability.

In this context, Serverless Computing is emerging as a cloud model that completely abstracts infrastructure complexity for end customers by offering high-level interfaces and fully automated management of services [1]. Two of the most diffused models of Serverless computing are FaaS and BaaS. The first offers event-driven execution of customer code encapsulated in elements called functions, while the second provides specialized and fully managed ad-hoc services, such as databases, object storage, and messaging middleware [2].

The joint use of the FaaS and BaaS models is a common pattern adopted in cloud-native applications [3]. It enables the definition of service business logic in FaaS functions and the use of managed and pre-configured services of the BaaS layer to solve complex problems. However, this integration is hindered by the absence of standards, heterogeneity in services and protocols, and lack of infrastructure support, often relying on a direct integration of BaaS service inside function code. Direct integration of service in function code couples it with a specific BaaS service, reducing code modularity and reusability and increasing vendor lock-in on service implementations.

To overcome these problems, we propose the Serverless Function Inversion of Control (SFIOC) platform. SFIOC extends existing FaaS architecture, enabling the integration of BaaS services into FaaS functions through Dependency Injection (DI). Dependency Injection (DI) is a specific implementation of the Inversion of Control (IoC) pattern where object dependencies are provided to it, rather than the object creating them itself. SFIOC proposes a novel infrastructure support that enables the dynamic and at-runtime resolution of service dependencies that are injected into function runtime. This promotes loose coupling and enhances testability by allowing dependencies to be easily swapped out based on execution context. SFIOC further advances state of the art by proposing an architecture extension of existing FaaS components to reduce runtime overhead introduced by the dynamic resolution of dependencies. Capabilities of SFIOC are demonstrated by an extensive testbed conducted over both public cloud and private hosting solutions by extending two commercial FaaS offers. The distributed and heterogeneous nature of this testbed is representative of a possible real Cloud Continuum deployment.

II. Background and Related Works

FaaS is a Serverless general-purpose cloud computing model that executes customer-defined units of code, called *functions*, when triggered by an incoming *event*. The customer specifies through the definition of a *workflow* the association between the event and corresponding functions executed.

A typical FaaS architecture can be summarized into three main architecture components depicted in Figure 1: the *trigger*, which receives input from the external world and translates them into FaaS manageable events, the Controller which receives events from triggers and routes events to designed functions configured by workflow and ensures that enough

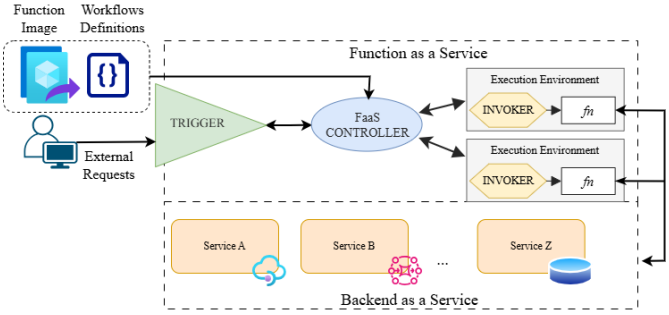

Fig. 1: FaaS conceptual architecture showing trigger, controller, and invoker as main architecture elements

resource are assigned for event processing, and the invoker that receives events from the controller and manages the instantiation and lifecycle of the function.

A. Service integration patterns in FaaS model

The FaaS is an intrinsically general-purpose, stateless, event-driven model. For these characteristics, it often relies on external services for tasks, such as data storage, synchronization, and message passing. These services, when provided with the same characteristics of total transparency of the infrastructure, simplified management, and a pay-per-consume model [4], belong to a second Serverless Computing model called Backend as a Service (BaaS). Indeed, over the past years and way before the diffusion of FaaS, cloud providers have enriched their offerings by providing an ever-increasing number of managed and already configured services, such as databases, storage, and message brokers enriching their BaaS offer.

Integration of BaaS services into FaaS function is a frequent pattern exploited in the community as it enables to leverage qualities of the Serverless model in the entire cycle of development of a service [5].

BaaS integration in FaaS can undergo two main approaches, which encompass respectively the call of the BaaS service inside the workflow management process or directly in the function code.

In the first approach, workflows are extended to support the definition of task. This transforms workflows in procedures that are executed by the FaaS controller or an external service and enables the integration of calls to service and functions in a unique event-driven flux. Examples of these approaches are AWS Step Function [6] and the Cloud Native Computing Foundation Serverless Workflows [7] that enables specifying calls to services and function triggering as composition of tasks supporting also iteration, branches, and retry logic. As a drawback, this approach requires the creation of a centralized controller to orchestrate and gather results of the different steps with an additional overhead for the infrastructure and limited scalability. Moreover, these technologies are often highly tailored to a single cloud provider and hardly scale over cloud continuum environments encompassing multiple vendors and hosting models [8].

B. Service Integration in FaaS function

The function integration approach, featuring direct integration of services inside function code, presents a lower ingress barrier for developers who can program integrations as they normally do with classic service development.

However, integration from within function business logic presents non-negligible performance concerns caused by the intrinsic nature of FaaS functions. The short lifespan of function instances, their transparency in location execution, and the absence of native mechanisms to retain the function state hamper the application of optimization patterns widely used in service and microservice service integration [9].

To overcome this limitation, many approaches have been proposed in both research and commercial platforms exploiting longer-living functions [10], specialized layers optimizing access to services [5], [11], [4], or tailored services natively integrated in FaaS platform [4]. However, these approaches tend to sacrifice function modularity, reusability, maintainability, and infrastructure flexibility as they statically couple a function to a specific service implementation.

To overcome this issue, we propose the introduction of native support of the Dependency Injection (DI) pattern in Function-as-a-Service (FaaS) platforms. The DI pattern, part of the Inversion of Control (IoC) family, advocates for delegating control flow to external entities, typically through frameworks or containers, thereby inverting the traditional flow of control. This inversion empowers developers to decouple components, promote modular design, and enhance the flexibility and extensibility of software systems. Support for this pattern has been historically implemented mainly by application containers in Service-Oriented Architectures (SOA). However, with the progressive reduction in the size of deployed services, first with microservice architectures and then with FaaS functions, the complexity and coarse-grained nature of Application Containers has discouraged their adoption in modern deployments.

The use of the DI pattern in FaaS to decouple functions from dependent services has already been proposed in previous works [12], [11]. In both cases, the authors support injecting dependencies into functions through the provision of a Software Development Kit (SDK) to be used in functions. However, this approach limits the dynamicity of the DI pattern, preventing the dynamic replacement of services without reconfigurations or code changes. Moreover, loading a heavy SDK in functions can result in performance degradation during the loading and execution of functions [13].

Our solution, SFIOC proposes an architecture that extends existing FaaS platforms by natively supporting the injection of BaaS services into FaaS functions. The runtime resolution of dependencies enables the greatest level of extensibility and modularity, allowing dynamic changes to dependencies without code modifications. SFIOC also reduces the overhead associated with DI by supporting runtime and asynchronous resolution of dependencies in FaaS infrastructure components.

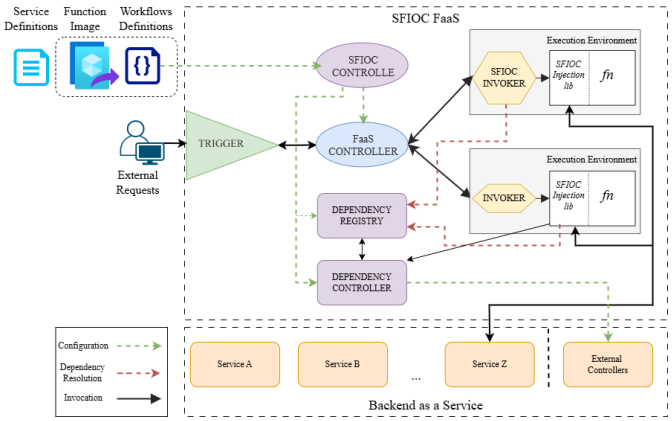

Fig. 2: High level architecture of our solution SFIOC featuring the SFIOC controller, the Dependency Registry, and the Dependency Controller enabling injection of manged BaaS service into functions

III. Serverless Function Inversion of Control (SFIOC)

Herein, we present our solution SFIOC, a platform that extends existing FaaS architecture to provide native support for DI into functions. SFIOC aims to provide a powerful support to easily integrate BaaS services into serverless workflows, facilitating DevOps operations from early prototyping to their optimization in production. We define *enrollment* as the actions of provisioning, configuration, and registration of a new BaaS service into SFIOC. Once a service is enrolled into a SFIOC platform, it can be dynamically injected at runtime as a dependency into one or multiple functions.

The architecture of SFIOC, as depicted in Figure 2, comprises three main elements: the SFIOC Controller serves as the central point for configuring DI-enabled workflows and orchestrating SFIOC components, the SFIOC Dependency Controller operating as a waypoint to enroll and provision BaaS services, and the SFIOC Dependency Registry which stores enrolled BaaS services and the metadata to support dependency injection.

In particular, the *SFIOC Controller* exposes custom APIs to enable the enrollment of BaaS services, the definition of new functions, and FaaS workflows. During the normal operations of the SFIOC platform, the customer defines and submits *service definitions* to the SFIOC controller, which requests to the Dependency Controller the enrollment of the services and obtains necessary information to target the service instance. This information is then associated with the service definition inside the Dependency registry. To activate a new workflow, a customer submits to the *SFIOC Controller* its definitions and the function code associated with. The SFIOC controller extracts DI dependency and verifies their availability in the Dependency Registry. If present, configure and activate the workflow by interacting with the FaaS controller (Figure. 2); otherwise, it starts a periodic job to check the presence of missing dependency in the system. If at any point in time, a new service configured met the dependency of the workflow, this is activated. This mechanism prevents the execution of workflows with unmet dependencies in the system. Finally, the SFIOC controller manages the lifecycle of other SFIOC architecture components, ensuring their reliability through failover and scaling actions.

The *SFIOC Dependency Controller* abstracts from specific BaaS providers Application Programming Interface (API)s to provide a unified provisioning and management support. A unified endpoint for the management of BaaS service reduces the overhead associated with heterogeneous interfaces and the vendor lock-ins faced during BaaS service integration. When a new service is *enrolled* in the system, the *Dependency Controller* communicates with the cloud provider API to gain access, configure the service, and retrieve essential interaction details, such as authentication tokens and endpoints. After a completed enrollment, the Dependency Controller starts a monitoring process to check it, eventually signaling its problems or unavailability to trigger mitigation strategies, such as substitution of the dependency or deactivation of interested workflows

Information to make requests to services is associated with service definitions and stored in the *Dependency Registry*. The Dependency Registry is the centralized repository that resolves dependencies for functions through an advanced query interface. It stores detailed service definitions, links them with service pointers, and provides necessary interaction details, such as endpoints and credentials, to enable integration and communication of services into functions.

To enable service DI into FaaS functions, our proposal provides the *SFIOC Injection Library*. This library, when imported into customer functions as a dependency, transparently implements the business logic necessary for service injection. Tailored for each supported language, the library reads the runtime configuration to resolve the dependencies that need to be injected. It provides interfaces that allow functions to interact with the injected services and interacts with the SFIOC Registry to retrieve the necessary service definitions and configurations. Additionally, the SFIOC Injection Library integrates the logic to send statistics over service calls to the Dependency Controller for monitoring and optimization purposes.

FaaS functions are architecture components with a short lifespan and limited knowledge of the execution context. The lifecycle of a function in a FaaS platform is unpredictable and can span only the processing time of a single event. Moreover, two subsequent invocations of the same function are not guaranteed to be executed in the same execution environment or physical node, preventing optimizations based on the execution context. These characteristics inherent to the FaaS model force our SFIOC Injection Library to be instantiated at each function invocation, preventing platform optimizations based on context-aware execution. To address these challenges, our solution proposes the *SFIOC Invoker*, which extends the FaaS invoker component by supporting the resolution of dependencies at the infrastructure level. Invokers

are long-living components in FaaS architectures that have access to contextual execution information and can maintain states or connections across multiple function invocations. The SFIOC Invoker enables a more efficient and ahead-of-time resolution of function dependencies, which are passed to the function at its startup. Moreover, when a function is handled by a SFIOC Invoker, the Injection Library functions only as an abstracting interface to access query services. This allows for loading a minimal version of the library at function startup, thereby reducing the function loading overhead.

To summarize, during the workflow triggering, requests from the external world follow a traditional FaaS platform flow, reaching at first the trigger where they are converted into events and passed to the controller, which will distribute event processing among available invokers. Here, if the targeted invoker is a SFIOC invoker, it invokes the execution of the function by passing the event to be processed and the dependency already resolved. The dependency resolution query is processed asynchronously by the SFIOC invoker so as not to impact FaaS workflow execution performance. In the case of a platform vanilla invoker, the execution environment first triggers dependency resolution in the SFIOC Injection library and then executes the code of the customer-defined functions. Finally, the library injects the dependency in the function code every time it is requested by the program.

A. SFIOC Implementation

The DI pattern has been extensively used to decouple business logic from various cross-cutting concerns, such as service composition, monitoring, access to storage, and authorization. Without loss of generality, we now present the first implementation of SFIOC, which enables the dynamic injection of Access Control List (ACL) BaaS services into functions.

We developed the initial prototype of SFIOC as a modular solution that can be extended and specialized for hosting on major public and private FaaS platforms backed by heterogeneous BaaS services. Currently, the specializations include Azure Functions and the Knative platform (Figure 3), representing, respectively, a widely adopted public solution and a private one.

The SFIOC Controller implementation follows the operator pattern, exposing an OpenAPI-compliant interface for defining SFIOC services, functions, and workflows. The controller implements a control loop process that continuously monitors the state and configuration of the infrastructure, executing reconciliation strategies to achieve the desired state. In on-premise environments, where infrastructure control is more accessible, this process includes the configuration and lifecycle management of SFIOC components.

The Dependency Registry is implemented using existing document database solutions with rich query interfaces. The decision not to mask specific database interaction protocols with a standardized API service is driven by the performance overhead such a service would introduce in the dependency resolution chain. In the future, as more FaaS platforms are

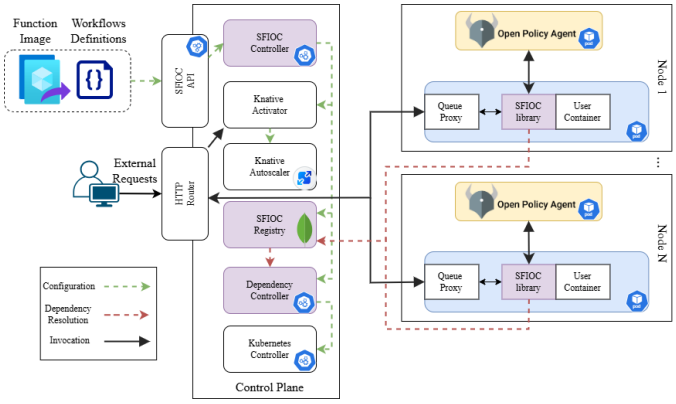

Fig. 3: Representation of SFIOC implementation extending the Knative platform featuring both Library-based and Invoker-based injections

supported, other strategies may be considered. This factor is, however, mitigated by the choice of databases that support the same MongoDB client library. Specifically, Azure deployments use Cosmos DB, while Knative deployments use a self-hosted instance of MongoDB.

The Dependency Controller is implemented as a series of specialized controllers that expose a uniform interface for BaaS service management, tailored to each serverless provider considered. These controllers follow the operator pattern, aiming to remap SFIOC controller configurations onto BaaS services, extract access information, and monitor their availability. On the Knative platform, this behavior is realized by implementing a controller that manages the provisioning and lifecycle of an Open Policy Agent (OPA) deployment (Figure 3).

We developed the SFIOC Injection Library in three different languages: JavaScript, Java, and Go. These languages were chosen due to their widespread use in the community and their representation of different FaaS function performance profiles [13]. The libraries expose an *injector* interface, which is passed as an argument at the startup of the function. Through this interface, functions can obtain the necessary dependencies, cast them if needed, and interact with the services, as shown in Listing 1.

Listing 1: Pseudo code example of a function that takes a triggering event and an injector endpoint as invocation arguments and queries the ACL service obtained via the dependency injection endpoint.

```
function main(event, Injector in)
{
  service (ACLService) = in.inject("ACL")
  service.authorize(...)
  ...
}
```

Finally, the SFIOC Invoker, which is considered only for the Knative platform, is developed as an extension of Knative Serving. This process reads configurations set by the SFIOC

Controller in the function deployment descriptor to know which dependencies to resolve for each function. It then starts a periodic asynchronous fetch task that queries the Dependency Registry to resolve dependencies. The results of this task are stored in the memory of the SFIOC Invoker and passed to the function as invocation arguments at each event arrival.

IV. Experimental Results

Herein, we assess the capabilities of our proposal in enabling dynamic dependency injection of BaaS services into FaaS functions while hosted on different FaaS environments characterized by varying resource availability and the performance implications of the architectural choices adopted.

The hosting environments were chosen to represent two different scenarios of the Cloud Continuum: a public large-scale cloud provider and a private Edge Computing deployment. The testbed consists of two separate environments: one hosted on the public cloud offering of Microsoft Azure, and the other extending a dedicated self-hosted Knative cluster running on three Virtual Machines, each equipped with 4 VCPUs, 8GB of RAM, and 120GB of disk space.

In the Azure testbed, we deployed the SFIOC Controller and Dependency Controller components as Azure Container Apps, while the SFIOC Registry is hosted through the BaaS service Azure Cosmos DB. The BaaS service considered for injection into functions is a deployment of the open-source OPA service hosted inside an Azure Container App. Notably, the entire deployment of SFIOC is feasible through serverless solutions, simplifying the deployment and management of the platforms.

In the Knative testbed, representing an Edge Deployment with two worker nodes, the SFIOC Controller and Dependency Controller are deployed as Knative Services, empowered with the necessary access rights to control cluster resources [14]. The SFIOC Registry is embodied by a MongoDB database, and the OPA service operates as Kubernetes Services. Finally, both vanilla and SFIOC invokers are deployed.

The testbed is organized into two sections: the first demonstrates the capabilities and evaluates the performance of SFIOC in injecting dependencies at runtime into FaaS functions, while the second evaluates the performance improvements achieved by adopting the SFIOC Invoker compared to the SFIOC Injection Library and a widely adopted service integration pattern.

A. SFIOC Library Dependency Injection Performance

In the first testbed, we demonstrated the capability of SFIOC to dynamically inject a service into functions across the two platforms considered. We measured the performance overhead introduced by the task of dependency resolution. The function tested accesses the ACL service injected through an HTTP request, verifying the access rights of a user passed as an argument in the triggering event. The function was developed in the three languages currently supported by SFIOC and associated with an HTTP trigger endpoint through workflow definitions, resulting in a total of six workflows: three on the self-hosted Knative platform and three on the Azure platform. We submitted a constant number of 10 requests per second for 5 minutes to each workflow and gathered execution statistics.

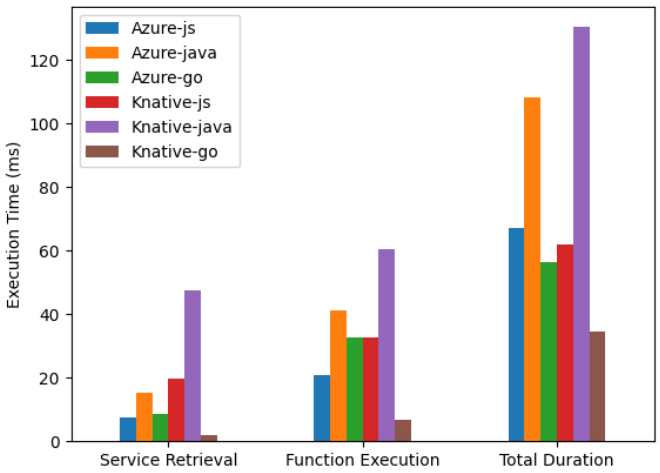

Fig. 4: Time taken by the platform to retrieve the service and execute the function compared to the total execution time in the six workflows when a constant request rate of 10 requests per second is submitted.

Figure 4 illustrates the average execution time for service retrieval, function execution, and total duration across the six workflows considered. The results reveal heterogeneous behavior across different environments and library implementations. Notably, functions implemented in Go consistently outperformed those in other languages across all tasks and hosting platforms. Service retrieval introduced a limited but very variable overhead, averaging 20%, with a minimum of 6% for the Go function hosted on Azure and a maximum of 36% for the Java function hosted on Knative. These variations are significantly influenced by the implementation choices of the executing frameworks and libraries used to query the Dependency Registry service, emphasizing the importance of carefully selecting development frameworks based on the application scenario. These results also underline the importance of architecture solutions, such as our proposed SFIOC Invoker, offloading service integration tasks from function to infrastructure.

B. Infrastructure performance Optimization

In the second testbed, we evaluate the performance of dependency injection when directly offered by the infrastructure. Specifically, we assess how the platform behaves when the resolution of the dependency to be injected into the function is handled by either the Injection Library or the SFIOC Invoker. For reference, these results are compared with one of the most common service integration patterns: the API Gateway, where the HTTP Router (Figure. 3) queries the authorization service to check access rights before invoking the function. The function tested is the go version tested in the previous

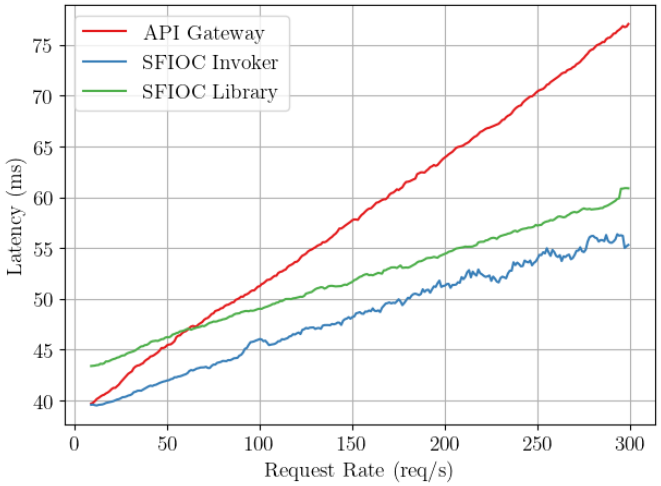

Fig. 5: Function execution duration when service integration is implemented via the API Gateway, SFIOC Library, and SFIOC Invoker, with a constantly increasing number of requests from 0 to 300 requests per second

testbed. We submitted a linearly increasing number of requests, ranging from 0 to 300, to the three workflows, reaching the resource saturation of the self-hosted Knative environment. This approach allows us to tests the behavior of the platform under different loading conditions.

Results shown in Figure 5 indicate that at low request frequencies, both the API Gateway pattern and the SFIOC Invoker strategies deliver optimal performance, highlighting the benefits of infrastructure-supported service integration. However, the performance of the API Gateway degrades at higher request frequencies due to the increased complexity of scaling this component, which must handle incoming requests while verifying authorization for each one. In contrast, the two solutions leveraging SFIOC-based service integration, benefiting from the intrinsic fine-grained scalability of the FaaS model, scale more gracefully. Specifically, dependency injection resolved by the invoker consistently provides a significant performance improvement of 4 ms on average per request, compared to the total overhead of 6 ms on average introduced by the DI evaluated in the first testbed. This test underscores the importance of architectural optimizations, particularly in constrained execution environments such as edge computing. These optimizations offer advantages for customers in terms of reduced end-to-end latency and for cloud providers by enhancing infrastructure throughput under the same operating conditions.

V. Conclusion and Future Works

In conclusion, the integration of Function as a Service (FaaS) and Backend as a Service (BaaS) models in serverless cloud computing is highly desirable for its potential to enhance modularity and flexibility. However, current solutions lack effective support for seamless integration, leading to challenges, such as reduced code modularity and increased vendor lock-in. This paper presents SFIOC, a platform that extends existing FaaS architectures to enable dynamic dependency injection of BaaS services into functions. Our experimental results demonstrate the effectiveness of SFIOC in injecting dependencies at runtime and optimizing performance through the SFIOC Invoker. We believe that the integration of Dependency Injection in FaaS platform could have a huge potential, especially when considering Cloud Continuum execution environments [3] in which the integration with BaaS service becomes more challenging for the increased heterogeneity in service. In future work, we plan to expand the capabilities of SFIOC, supporting more FaaS platforms and enhancing performance in service calls.

VI. Acknowledgments

The European Union supported this work under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, “SEcurity and RIghts In the CyberSpace” (PE00000014 - program “SERICS”) CUP:J33C22002810001.

References

[1] S. Risco, G. Moltó, D. M. Naranjo, and I. Blanquer, “Serverless Workflows for Containerised Applications in the Cloud Continuum,” *Journal of Grid Computing*, vol. 19, pp. 1–18, 9 2021.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica, and D. Patterson, “Cloud Programming Simplified: A Berkeley View on Serverless Computing,” 1 2019.

[3] P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski, “Hybrid serverless computing: Opportunities and challenges,” *Serverless Computing: Principles and Paradigms*, pp. 43–77, 2023.

[4] H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless computing: A survey of opportunities, challenges, and applications,” *ACM Comput. Surv.*, vol. 54, Nov. 2022.

[5] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, “The serverless computing survey: A technical primer for design architecture,” *ACM Comput. Surv.*, vol. 54, Sept. 2022.

[6] J. P. Buddha and R. Beesetty, *The Definitive Guide to AWS Application Integration: With Amazon SQS, SNS, SWF and Step Functions*. Apress, 2019.

[7] “Serverless workflow,” 2024.

[8] A. Mathew, V. Andrikopoulos, F. J. Blaauw, and D. Karastoyanova, “Pattern-based serverless data processing pipelines for function-as-a-service orchestration systems,” *Future Generation Computer Systems*, vol. 154, pp. 87–100, 2024.

[9] R. Basu Roy, T. Patel, R. Liew, Y. N. Babuji, R. Chard, and D. Tiwari, “Propack: Executing concurrent serverless functions faster and cheaper,” in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC ’23, (New York, NY, USA), p. 211–224, Association for Computing Machinery, 2023.

[10] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, “Durable functions: Semantics for stateful serverless,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[11] T. Larcher, P. Gritsch, S. Nastic, and S. Ristov, “Baasless: Backend-as-a-service (baas)-enabled workflows in federated serverless infrastructures,” *IEEE Transactions on Cloud Computing*, pp. 1–15, 2024.

[12] 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, (New York, NY, USA), p. 23–28, Association for Computing Machinery, 2021.

[13] N. Mahmoudi and H. Khazaei, “Performance modeling of serverless computing platforms,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2834–2847, 2022.

[14] D. O. Roles, “How kubernetes changes operations,” *Operating Systems and Sysadmin*, p. 36.