SN Computer Science (2024) 5:678
https://doi.org/10.1007/s42979-024-03022-w

ORIGINAL RESEARCH

Self-Provisioning Infrastructures for the Next Generation Serverless Computing

Stefan Nastic[1]

Received: 21 October 2023 / Accepted: 31 May 2024

Abstract

Serverless computing has ushered in a transformative paradigm, with a promise to alleviate developers from the intricacies of infrastructure management. However, current serverless platforms predominantly offer only serverless compute capabilities. As a consequence, the application developers are once again tasked to explicitly provision and manage the backend services (BaaS), such as object stores or API gateways, the infrastructure, and the configuration models. This violates the main promise of serverless computing and erases much of the practical benefits of the serverless paradigm. It also introduces the challenges of managing the application execution environment, which includes maintaining provisioning and deployment scripts, configuring and managing access permissions, and scaling the services during runtime. To address these challenges, in this paper we introduce a novel paradigm for the next generation of serverless computing, called self-provisioning infrastructure. The self-provisioning infrastructure is an infrastructure that is capable to automatically and autonomously (with zero-configuration and zero-touch) provision serverless functions, their infrastructure, and their supporting BaaS services. To achieve this vision, we introduce novel design principles, models, and mechanisms that are formalized via *novel programming, function, and system models*. With this novel paradigm, we intend to fortify the core design principles of serverless computing but also extend them *to the entire application execution environment*. By doing so, we aim to enable the next-generation serverless computing in the Edge-Cloud continuum.

Keywords Self-provisioning · Serverless computing · Infrastructure provisioning · FaaS · Edge-cloud continuum

Introduction

Serverless computing enables a new way to build and scale applications by allowing developers to decompose traditionally monolithic applications into finer-grained "triggers" (events) and "actions" (FaaS functions). Serverless platforms typically provide a hosting and execution environment for such functions [1, 2]. Developers write the function business, logic while the serverless platform performs the notoriously tedious and complex tasks of provisioning, scaling, and managing the backend compute resources [3].

The emergence of serverless computing has ushered in a transformative paradigm, with a promise to alleviate developers from the intricacies of infrastructure management [1, 4–6]. Current serverless platforms typically provide only serverless compute capabilities and application developers still need to explicitly deal with the supporting backend services such as object store, API gateway, secrets manager, or in-memory cache. These services are known as Backend-as-a-Service or BaaS services [1] and they are increasingly an integral part of many serverless applications [7–10]. By being a part of the software stack, such services reimpose the intricacies of backend service management to developers, who again need to explicitly worry about deploying and managing application backend services. Consequently, we face new challenges to fortify the core promise of serverless computing, that is, writing FaaS functions without worrying about the complexities of infrastructure provisioning and management, while at the same time seamlessly extending the serverless promise to encompass provisioning and management of the BaaS services but also the rest of the infrastructure. The matter is further complicated by the increasing need to run such services and applications across the entire Edge-Cloud Continuum [11–13].

✉ Stefan Nastic
snastic@dsg.tuwien.ac.at

[1] Distributed Systems Group, TU Wien, Argentinierstraße 8, 1040 Vienna, Austria

Published online: 26 June 2024

Recently, federated serverless computing [14–19], sky computing [20], hybrid serverless computing [21] and serverless edge-cloud continuum [12] have been gaining in popularity due to numerous benefits they can bring to the next-generation serverless workflows and applications. Such applications are typically deployed across multiple cloud providers and edge devices to reduce cost [22], improve scalability [18, 23, 24] or increase serverless application resilience [25]. However, current serverless functions are still largely dependent on the specific cloud provider configurations and features, e.g., specific memory configuration, specific event source integration, and frequently specific features of the BaaS services. This inherent vendor lock-in, combined with a large number of (vendor-specific) cloud and edge BaaS services and their configuration options, makes it very difficult in practice to reap the benefits of the federated edge and cloud serverless infrastructures. Current serverless computing models support only static configurations per function, e.g., as memory and CPU. Once specified, these configurations remain the same for all subsequent function invocations. Such static configuration and resource allocation models do not yield optimal resource utilization and typically lead to over- or under-provisioning of resources [26–28]. There is still limited support for dynamic configuration models that are applied per function instance and that can account for the variations in, e.g., input data size and hardware heterogeneity. Consequently, this gives rise to multiple challenges that impact provisioning and managing infrastructures for the next-generation serverless applications in the Edge-Cloud continuum.

Provisioning execution environment for serverless functions and applications State of the art and its limitations. Contemporary serverless platforms and approaches are focusing mainly on offering serverless compute capabilities, meaning that the users are relieved from explicitly provisioning the compute resources (but some configuration is still required). However, the rest of the *application's execution environment*, including its infrastructure, BaaS services, and configuration and resource allocation models still need to be explicitly provisioned by the users.

Infrastructure as Code (IaC) is currently considered the best practice for provisioning and managing application infrastructure in the cloud. It relies on declarative configuration languages [29–33] or more recently general purpose programming languages [34, 35] to capture infrastructure resources, their interdependencies, and configurations. These approaches introduce numerous benefits to infrastructure management by facilitating process automation, guaranteeing consistency, and increasing resilience and cost savings. However, by design, they still require users to explicitly and largely manually write the infrastructure provisioning logic. This breaks the core promises and design principles of serverless computing but also results in: (i) code duplication, since the same concepts, such as S3 bucket need to be defined both as application code and IaC configuration, (ii) vendor-specific limitations, since writing control-plane-specific provisioning instructions is often required, and (iii) a mismatch between declarative configurations and dynamic infrastructures.

Faasifiers are special infrastructure management and deployment solutions, which specifically target serverless computing. Faasification is a process of converting monolithic applications to run on serverless platform, either as stand-alone serverless functions [36–38] or hybrid applications, which offload parts of the code as serverless functions [25], while retaining the same interface to the user [23]. Despite the benefits they provide in terms of simplified provisioning, deployment, and management of FaaS functions, the faasifiers focus on facilitating the serverless functions and they largely neglect the supporting BaaS services, the infrastructure, and the configuration and resource allocation models.

Clearly, there is still insufficient support to enable the provisioning of infrastructure in a purely serverless fashion. This leads us to four main research questions, which motivate our research efforts toward self-provisioning infrastructures:

1. Can we extend the core serverless principle to the entire application's execution environment in the next generation of serverless computing?
2. How can self-provisioning of BaaS services be enabled?
3. How can self-optimization of configuration and resource allocation models be achieved?
4. Can we effectively account for non-functional concerns and SLO awareness in the next generation of serverless computing?

Contributions In this paper, we introduce a novel paradigm called *self-provisioning infrastructure (SPI)* for the next generation of serverless computing. The self-provisioning infrastructure is an infrastructure that is capable of automatically and autonomously provisioning the execution environment and application runtime for serverless functions. With this novel paradigm, we intend to fortify the core design principles of serverless computing and extend them *beyond the serverless compute to encompass the entire application execution environment*. By doing so, our vision is to enable the next-generation serverless computing in the Edge-Cloud continuum. To this end, we introduce novel approaches and models of the self-provisioning infrastructure that are formalized via novel (i) Programming model, (ii) Function model, and (iii) System model. The SPI models specify novel abstractions, such as *serverless primitives and runtime mixins*. The serverless primitives abstract the underlying

infrastructure capabilities (such as storage attachments) enabling pure utility-based consumption of the infrastructure resources at fine granularity. Together with the *Function and BaaS models* they bring portability and interoperability at multiple levels including business logic level, platform level, and serverless workflow level, which are some of the key preconditions to achieve the next-generation serverless computing in federated infrastructures. The runtime mixins introduce the SLO-awareness to the self-provisioning infrastructures, by enabling self-provisioning of the non-functional concerns such as reliability, but also self-instrumentation, self-auditing, self-updating, and self-governance. The SPI introduces novel *AI-based* runtime mechanisms, which enable self-optimization of infrastructure's configuration models. This warrants an infrastructure that is optimized for performance and efficiency while also being cost-aware.

Paper structure The rest of the paper is structured as follows. "Background and Related Work" section discusses the state-of-the-art approaches in comparison with our approach. In "Principles and Models of Self-Provisioning Infrastructure" section , we present the main design principles, models, and mechanisms of the self-provisioning infrastructure. "Designing and Implementing Self-Provisioning Mechanisms" section introduces the most important self-provisioning mechanisms and discusses in detail their design. In "Discussion" section, we discuss the main trade-offs and practical applicability of the self-provisioning infrastructures. Finally, "Conclusion" section concludes the paper and gives an outlook on future research.

Background and Related Work

Infrastructure as Code Approaches

There have been numerous research and commercial IaC approaches and they can be broadly classified into first- and second-generation IaC approaches. Examples of the first-generation IaC approaches include: Terraform [29], CloudFormation [30], TOSCA [31], Serverless framework [32], and Ansible [33]. Such approaches typically enable defining the infrastructure and its dependencies, by using custom directives, definitions, or a declarative configuration language, which are typically specified as YAML files. The second-generation IaC approaches include AWS CDK [34] and Pulumi [35]. Instead of using a custom configuration language, as most of the first-generation IaC tools do, these approaches leverage existing language ecosystems to define, deploy, and manage cloud infrastructures. This allows for more flexible and modular infrastructure code while abstracting away the boilerplate functionality. All of these approaches require the user to explicitly write the provisioning logic in addition to the application business logic. This is a significant limitation since it results in a lot of code duplication as facets are expressed both as application code and IaC configurations. The declarative nature of IaC approaches, which are used to describe static infrastructures is not well suited for the dynamic nature of serverless infrastructures and their configuration and resource allocation models. In relation to our self-provisioning infrastructures, these approaches are enablers, because they can be used to capture the core infrastructure provisioning logic.

Several approaches that offer specialized frameworks for provisioning and deploying serverless applications recently emerged. These include Kotless [39], AWS Chalice [40], Zappa [41], and Osiris [42], as well as various so-called FaaSifiers [23, 43]. For example, AWS Chalice is a tool for Python developed by AWS. The tool uses its own DSL to define HTTP APIs and event handlers, which then get automatically provisioned on AWS, making it easy to create simple serverless applications. Osiris is a tool for Kotlin which also provides a custom DSL for defining HTTP API events. Osiris does not interact with AWS services directly, instead, it generates CloudFormation definitions. While such approaches share some conceptual similarities with self-provisioning infrastructure, they mainly require using their custom DSLs, which steepens the learning curve. Additionally, they usually target a single cloud provider and they have limited interoperability and portability considerations.

There are also approaches to serverless object stores (e.g., S3), serverless rational databases such as Amazon Aurora[1] and serverless key-value stores such as DynamoDB.[2] While these BaaS services are provided as serverless offerings, they still need to be explicitly maintained (together with their configuration models, access policies, etc.) by the users as a part of the software stack that supports the application execution environment. However, these are mature technologies and self-provisioning infrastructure can seamlessly integrate with such approaches in a similar manner as they do with traditional BaaS services.

Configuration Optimization and Tuning Approaches for Serverless Functions

Various approaches attempt to tune the resource configuration of serverless functions to ensure that the serverless applications meet their SLOs while minimizing the costs. The best practice is to build a performance model of the application's serverless functions and use that model to tune the functions' configuration models. These approaches can be divided into two main categories: (i) approaches that

[1] https://aws.amazon.com/rds/aurora/.

[2] https://aws.amazon.com/dynamodb/.

require a-priori profiling of functions to build a performance model in an offline fashion and (ii) approaches that monitor function executions during runtime to build the performance model in an online fashion.

Systems that rely on the a-priori profiling typically execute individual functions under varying resource configurations with typical input data to learn about their performance. The resulting performance profile is used to configure the function's resources in production in order to meet the response time SLO and/or cost requirements. These approaches can be further classified by the algorithm used to determine the resource configurations. Most systems that find suitable function configurations for an entire workflow rely on graph algorithms [44–46] or use a max-heap [10]. To optimize a single function or job, linear, binary, and gradient descent search [47], Bayesian Optimization [48], and CPU time accounting [49] have been used. A common drawback of the a-priori profiling systems is that a "typical workload" needs to be defined for each function, which might not be possible for functions that have highly variable inputs.

Systems that build the performance model in an online fashion either rely on historical or live monitoring data. Some approaches passively monitor execution [47, 50, 51], while others assign different configurations to the function runs until the performance model is complete [52, 53]. The latter commonly use statistical methods, such as Bayesian optimization, to speed up this process and reduce the number of configurations that need to be explored. However, by design, these approaches require the allocation of additional resources for collecting and processing the monitoring information during the entire application lifetime. This usually leads to a suboptimal total cost profile of serverless applications.

AI-Based Approaches for the Next-Generation Serverless Computing

As the complexity of edge-cloud environments is growing it becomes a great target to apply AI and ML techniques to not only do various optimizations but also to provide specialized tools to simplify the development and provisioning of the next-generation serverless applications. For example, recent trends that focus on building intelligent tools that can help developers [54], help discover vulnerabilities [55], assist with bug fixing [56], and optimizing configurations [27], as well as provide easy-to-understand recommendations based on monitoring data available. These techniques can serve as building blocks to achieve the next generation of severeness computing and naturally complement our self-provisioning infrastructure. Other approaches use AI and ML to gain a better understanding of serverless application operations by extending existing serverless observability solutions [57], AI for IT operations (AIOps) [58–60] and related areas, which again naturally complement the self-provisioning infrastructure. Therefore, all of these approaches are enablers of the self-provisioning infrastructures as they can help further advance the SPI's runtime support.

Principles and Models of Self-Provisioning Infrastructure

The main design principle behind our approach is the concept of *self provisioning*. Self-provisioning denotes the ability of an infrastructure or a platform to **automatically and independently (with zero-configuration and zero-touch)** provision a complete execution environment for a serverless function solely from its business logic. For example, if a function requires an object store, a serverless platform should be capable of recognizing it, e.g., by detecting the use of 'botocore.s3.Bucket()' in the function's business logic. Moreover, the platform needs to also provision a suitable bucket, access policies, resource allocations, etc for the function.

It is generally accepted that serverless functions typically consist of a Function as a Service (FaaS) part and a Backend as a Service (BaaS) part [1]. Moreover, the FaaS part can typically be broken down into a trigger, a handler, and a runtime [45, 61]. The trigger defines a condition that causes an invocation event to be sent to the handler (e.g., the new file being uploaded to a bucket). The handler contains the actual business logic to process such an event (and platform-specific code, irrelevant to this discussion). Finally, the runtime consists of a language runtime (e.g., Java Runtime Environment) and any libraries specified as dependencies via, e.g., requirements.txt for Python or package.json for NodeJS functions. Even in this simplified FaaS model, we notice that the contemporary serverless platforms only self-provision the required runtime.[3] The remaining triggers, configuration models, BaaS services, etc need to be explicitly provisioned by the user.

The self-provisioning infrastructure (SPI) applies the self-provisioning paradigm to the **entire application execution environment**, including the application runtime, BaaS services, function resources, and so on. We argue that the *self-provisioning infrastructure is the next step in the evolution of serverless computing*. Moreover, it is a crucial step to enable extending serverless computing beyond a single cloud provider toward multi-cloud serverless computing, federated FaaS or so-called sky computing, but also beyond

[3] Some platform providers also provide support for simplified provisioning of ad-hoc features such as AWS Lambda's function URLs, but they still require the user to configure and provision them (https://docs.aws.amazon.com/lambda/latest/dg/lambda-urls.html).

the Cloud and toward the Edge in the so-called Edge-Cloud continuum [12].

Design Principles of Self-Provisioning Infrastructure

Self-provisioning is the main design principle behind the self-provisioning infrastructures. Other design principles of the self-provisioning infrastructure include:

1. *Self-optimization*—In addition to the self-provisioning principle, which requires automated and autonomous provisioning of the infrastructure, self-optimization states that the configuration models and resource allocations need to be optimized for performance, efficiency, and cost.
2. *SLO-awareness*—The self-provisioning infrastructure needs to account for the non-functional requirements of the application. It should be possible to specify such requirements as high-level SLOs, such as availability, rather than delving into low-level infrastructure provisioning requirements, such as CPU capacity or memory allocation.
3. *Utility-based consumption*—The infrastructure should present itself as a transparent utility, letting developers focus only on application business logic. The developers should be shielded from having to explicitly deal with the infrastructure details. The infrastructure must deliver its capabilities transparently and it must possess the capability to adapt to the changing needs of the applications.
4. *Cost-awareness*—All self-provisioning mechanisms within the SPI should possess an intrinsic cost sensitivity. Decisions enacted by these mechanisms, ranging from service provisioning to configuration tuning, should be implemented with an optimization goal of minimizing the costs.
5. *Fine-grained configuration and consumption*—Infrastructure components need to be accessible at various granularity levels. This requires precise calibration in how resources are provisioned, guaranteeing allocations that well align with application needs, while considering overall resource consumption. It also requires shifting from current per-function-type configuration models toward fine-grained per-function-invocation configuration models.
6. *Multi-level portability & interoperability*—To enable the next-gen serverless computing, in addition to the above design principles, the SPI needs to enable portability and interoperability at several levels: (i) business logic, i.e., function handler level, (ii) (virtual) platform, i.e., BaaS level, and (iii) application, i.e., serverless workflow level.

Models of Self-Provisioning Infrastructure

After providing a general overview and the main design principles of self-provisioning infrastructures, we next introduce the main concepts, techniques, and the formal model of self-provisioning infrastructures. Figure 1 gives a high-level overview of the SPI model. In the continuation, we mainly focus on discussing three main aspects of the SPI that are formalized via: (i) Programming model, (ii) Function model, and (iii) System model. To unlock the full potential of the SPI all of the models are required. However, it is also possible to adopt only a subset of the models and still receive benefits from the self-provisioning infrastructure paradigm.

Programming model The *Serverless primitives* provide the core APIs and protocols to the serverless functions to interact with its execution environment, including the BaaS services and the infrastructure. These primitives are represented by well-defined interfaces that enable the functions to interact with the underlying capabilities in an implementation-independent manner. An example of the serverless primitive is shown in Listing 1. The listing shows a partial interface for a key-value BaaS service, which exposes four functions, that can be used by a serverless function to manipulate the key-value pairs and to interact with the underlying key-value store. The exposed functions are *get, set, delete*, and *exists*. It is worth noting at this point, that we do not attempt to define an exhaustive list of the Serverless primitives, nor do we try to define standard interfaces for the primitives. There are already several attempts to achieve that including for example, [62, 63].

The Serverless primitives are platform- and vendor-independent and provide a high-level description of services that can be invoked by the serverless functions, similar to the operating system system calls. A set of such Serverless primitives can be seen as a virtual platform (or a virtual OS) on top of which we run the serverless functions. As discussed subsequently, the SPI "instantiates such a virtual platform" by provisioning concrete capabilities and capability providers. By enabling the developers to write serverless functions by using well-defined, generic serverless primitives instead of control-plain-specific or vendor-specific instructions, we can enable the SPI to "understand" the function's code or even the developer's intents.

This approach has an obvious drawback because in order to be general enough, it captures the least common denominator in terms of the functionality of the BaaS services. Its drawback is that the default SPI primitives can fail to capture all the features of various BaaS service implementations, e.g., Redis vs. Memcached. To remedy this, in addition to the default Serverless primitives, the SPI offers the possibility to the users to provide custom definitions of a serverless primitive and register it (together with its capability implementation) with the SPI. Consequently, the SPI also

considers such custom definitions while performing infrastructure self-provisioning.

As shown in Fig. 1, the SPI also provides abstractions *Runtime mixins* and *Function annotations*. Mixins are abstractions that enable creating specific units of functionality in isolation, which can be transparently mixed into some other functionality [64]. They are mainly used in object-oriented programming but can be useful as a convenient approach to provide additional lifecycle hooks for managed services or components. In SPI, the runtime mixins are used to enable the self-provisioning of mechanisms, which mainly address non-functional concerns, such as reliability, but also for self-instrumenting, self-auditing, self-updating, and self-governing.

```
/// A key-value interface that provides simple read and write operations.
interface readwrite {
  /// A key-value interface that provides simple read and write operations.
  use types.{bucket, error, incoming-value, key, outgoing-value}

  /// Get the value associated with the key in the bucket. It returns a incoming-value
  /// that can be consumed to get the value.
  ///
  /// If the key does not exist in the bucket, it returns an error.
  get: func(bucket: bucket, key: key) -> result<incoming-value, error>

  /// Set the value associated with the key in the bucket. If the key already
  /// exists in the bucket, it overwrites the value.
  ///
  /// If the key does not exist in the bucket, it creates a new key-value pair.
  /// If any other error occurs, it returns an error.
  set: func(bucket: bucket, key: key, outgoing: outgoing-value) -> result<_, error>

  /// Delete the key-value pair associated with the key in the bucket.
  ///
  /// If the key does not exist in the bucket, it returns an error.
  delete: func(bucket: bucket, key: key) -> result<_, error>

  /// Check if the key exists in the bucket.
  ///
  /// If the key does not exist in the bucket, it returns an error.
  exists: func(bucket: bucket, key: key) -> result<bool, error>
}
```

Listing 1 An example of a serverless primitive interface for key-value BaaS service (partial view)

An example of a runtime mixin is adding a retry mechanism to a serverless primitive or a more elaborate circuit breaker for improved reliability. When a function invokes a primitive the mixin acts as a form of a lifecycle hook, which is automatically invoked by the SPI. For example, the retry mixin is typically associated with the request post-processing phase, i.e., it is triggered if a serverless primitive returns an error instead of a result.

The function annotations enable the user to specify additional information that is useful or required by the SPI. For example, the annotations can be used to specify additional configurations for function triggers that cannot be automatically inferred from the SPI. The annotations can be applied on the function handlers or on the Serverless primitives. We discuss annotation usage in more detail in the following section.

System model Next, we discuss the most important components of the SPI's internal system model. Generally, the system model components are used to support other components in the SPI or to enable execution of the self-provisioning mechanisms. The main components of the SPI's system model are shown in Fig. 1 (right).

As we briefly mentioned earlier, the *Infrastructure capabilities* and the *Capability providers* are required to enable the Serverless primitives. The Infrastructure capabilities provide an abstract implementation of the Serverless primitives and they can be seen as a SPI internal representation

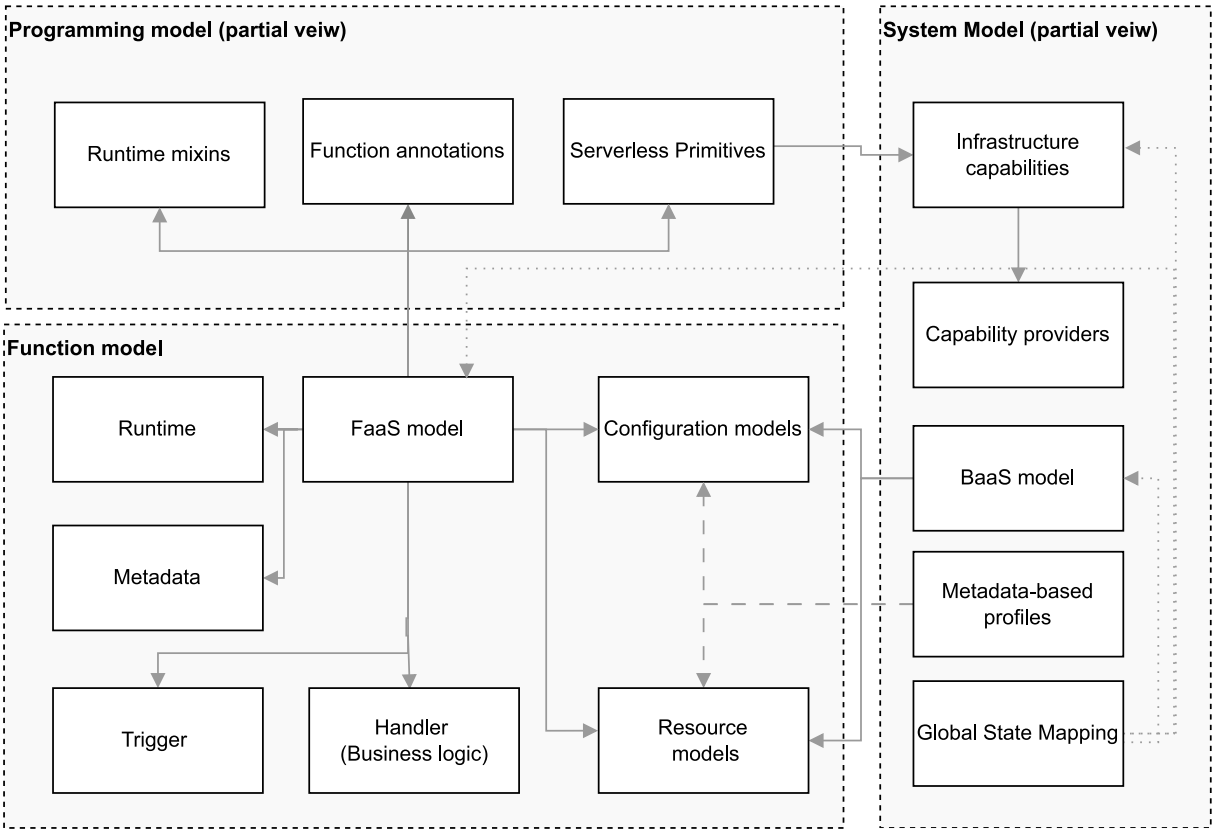

Fig. 1 Model of the self-provisioning infrastructures (partial view) [62]

of the Serverless primitives and their execution lifecycle. More specifically, each primitive has exactly one capability associated with it, in any given version of a self-provisioning infrastructure, at any given time.

Figure 2, show a simplified UML model of the Capability. The Infrastructure capabilities act as adapters between the Serverless primitives and the underlying Capability providers. This means, that their main role is to provide the boilerplate code needed to map the serverless primitive calls to the underlying Capability provider API call. The Capability providers are the actual services or components that expose the APIs and provide concrete implementations. Examples of the Capability providers are Redis, Memcached, and Dynamo DB. In addition to the request mapping, the Infrastructure capabilities also provide several *Lifecycle hooks* for extending the primitive invocation lifecycle. Most notably these include *preprocessor () and post-processor()* hooks, which enable the SPI to inject custom logic into the request pre- and post-processing phases. The Infrastructure capabilities also allow for registering *Runtime mixins* that enable adding custom lifestyle hooks. The capabilities also contain specific *Infrastructure provisioning code*. Such provisioning code can be created by a capability provider (similar to Kubernetes operators or Helm charts) or it can be auto-generated provisioning code using generative AI approaches such as problem-specific large language models [65].

The SPI relies on the *Binding configuration* to maintain the relationships between the primitives and the Infrastructure capabilities/Capability providers. This decoupling of the primitives from the supporting capabilities enables the self-provisioning infrastructure to completely separate the serverless functions (more precisely their business logic) from the underlying infrastructure. By doing so it also eliminates any business logic dependence on the underlying infrastructure, but also the dependence of the infrastructure on the function's business logic. This is one of the key preconditions for the SPI to perform the infrastructure provisioning fully by itself. With this approach, in addition to provisioning the infrastructure, the SPI also needs to make sure that there are infrastructure capabilities available that can adequately fulfill the referenced Serverless primitives. To achieve this, the SPI relies on an interface-based auto-wiring mechanism, which can automatically discover and "wire together" the primitives with their capabilities, based on the interfaces they expose/expect. It stores all the relevant information in the Binding configurations, but if needed those configurations can

also be customized explicitly by the user. Eliminating the dependencies between the functions and the infrastructure also brings benefits in terms of reduced vendor lock-in, increased portability, provider hot-swapping, and so on.

The remaining system components shown in Fig. 1 include *Infrastructure global state mapping* and *Metadata-based profiles*. They are best understood as helper modules, which are required to support self-provisioning mechanisms, as discussed in "Designing and Implementing Self-Provisioning Mechanisms" section.

Function model The self-provisioning infrastructure defines the Function model, based on the principle of the least common denominator of the available FaaS models and platforms [1, 4–6]. This is a deliberate design decision since we want the SPI to be compatible with the existing state-of-the-art serverless platforms and approaches. The SPI's function model is shown in Fig. 1 (bottom), and its main components include *Trigger, Handler, Runtime, Metadata, Configuration model and resource-allocation model*. Some of these components were already discussed at the beginning of this section. In short, the Trigger defines a condition that causes an invocation event to be sent to the handler. The Handler contains the function's business logic. The Runtime consists of a language runtime and function's dependencies. Provisioning and management of the Runtime can completely be delegated to the existing serverless platforms, as discussed earlier in the text.

The SPI explicitly considers the resource and configuration models, in order to enable their effective management. We refer to this as *self-optimization of configuration and resource-allocation models* a.k.a. configuration and resource tuning.

The SPI relies on an approach that uses the Metadata to determine the function's profile, which is then used to determine the baseline for the function's configuration and resource allocation. To this end, we build on our previous work on AI-based configuration tuning and metadata-based profiling [66]. The main idea of metadata profiling is to use apriori known information about a function to predict its dynamic runtime behavior. We refer to such information as static, apriori metadata and it is available at the provisioning time. Examples of such metadata include tags, user data, OS parameters, and so forth. Profile classifier (cf. Fig. 3 left) uses this metadata as input features to assign a suitable profile to a to-be-provisioned function. These profiles capture the dynamic characteristics and behaviors of the function and they are computed by the Profile generator (cf. Fig. 3 right). The details of this process are beyond the scope of this paper and we address them elsewhere [66].

The SPI uses metadata profiling and extends the profiles to include the function resource and configuration models. This effectively enables the SPI to derive a suitable resource and configuration model (e.g., function memory limits, its timeout, etc.) for a function, by only considering the function's metadata. This approach helps us address the so-called bootstrapping problem, inherent to many optimization techniques. This is the problem when the optimization technique cannot be applied until enough information is collected about a workload, such as function. Having long bootstrapping delays can lead to sub-optimal configurations and resource allocations.

After using the metadata-based profiling to establish an intelligent baseline for the configuration and resource-allocation models, the SPI can exploit state-of-the-art techniques to further optimize those models, such as reinforcement learning [67], linear, binary, and gradient descent search [47], Bayesian Optimization [48], and CPU time accounting [49]. Naturally, the SPI also allows the users to provide custom configuration models, in case they need to override the self-optimized values.

The *BaaS model* is the FaaS model's counterpart that represents the main abstractions and concepts of the BaaS services that are self-provisioned by the SPI, but due to space limitations, we refrain from further discussing it.

Designing and Implementing Self-Provisioning Mechanisms

Next, we look at the main classes of self-provisioning mechanisms and how such mechanisms are implemented. These mechanisms can be broadly classified into three groups: *(i) self-provisioning FaaS, (ii) self-provisioning BaaS and (iii) self-provisioning non-functional aspects, and iv) Self-optimization of configuration and resource-allocation models.* Main examples of *self-provisioning FaaS* mechanisms include self-provisioning of FaaS triggers, self-provisioning of FaaS programming models, and self-provisioning of application runtime. *Self-provisioning BaaS* includes, self-provisioning of storage attachments, key-value stores, ingress controllers, and persistent objects. *Self-provisioning non-functional aspects* is the broadest category and it includes mechanisms responsible for managing application reliability, elasticity, governance, performance, auditing, instrumentation, and so forth. *Self-optimization of configuration and resource-allocation models* (a.k.a. configuration and resource tuning) include self-provisioning of function or service resources such as memory, availability zones, IO throughput, min/max concurrency, pool size. Next, we discuss some of the most important examples in more detail.

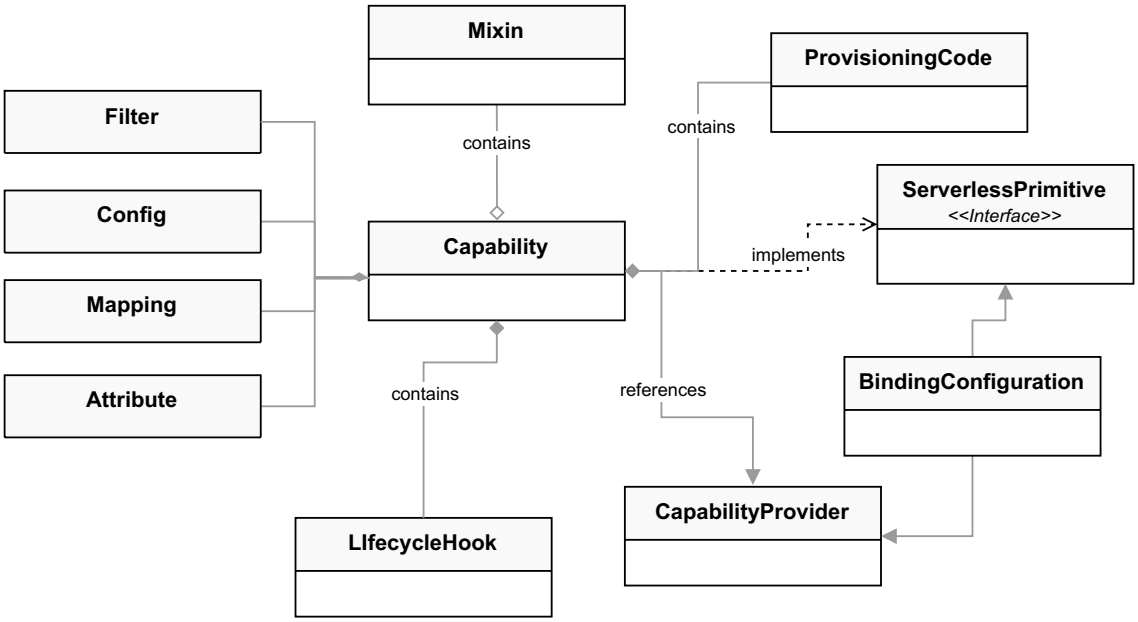

Fig. 2 Simplified UML diagram of capability structure

Self-Provisioning FaaS Triggers

Triggers are one of the key parts of the serverless application model. They help define when and how a specific serverless function is to be invoked. Examples of trigger events include an upload to an S3 bucket, an HTTP request from an API gateway, or a message added to a queue. However, configuring triggers can be a tedious and time-consuming task. More importantly, if done incorrectly, it can have detrimental consequences, e.g., due to a phenomenon known as a "runaway function". In such a situation an output produced as a result of a function is falsely recognized as a trigger event and an input for the same functions, resulting in an endless recursive loop of function invocations, potentially incurring significant costs [68]. This is a great example of a task that should be delegated to the self-provisioning infrastructure. Typically to configure a simple trigger, such as *"invoke a function when a new file with an extension '.png' is added to bucket 'uploads' "* one needs to interact with an object storage service, such as S3, create a new bucket and define its configuration model, including the access management. After that appropriate suffixes and prefixes need to be added, together with the event for adding the file to the bucket such as an HTTP verb (e.g., 'PUT'). Further, one has to configure the appropriate identity and access permissions to allow communication between the function and the bucket. Finally, an output bucket for storing the results such as a rescaled output image needs to be created. An alternative to the last step is to create a custom business logic, which makes sure that the recursive invocation is detected and prevented in order to prevent the runaway function.

The next generation of serverless computing should relieve users from such mundane and error-prone tasks. With the self-provisioning infrastructures, this issue is mitigated by taking advantage of the **Function annotations**, which can provide the necessary hints to the infrastructure (e.g., the file extension and the bucket name) on how it should self-provision the trigger. Other information such as the access policies and the need for an additional output bucket can be easily inferred by the infrastructure from the function's business logic and its interaction patterns with the storage. For example, if a function wants to store the same file type, the self-provisioning infrastructure can decide to self-provision another bucket for the results instead of storing them in the source bucket. With the SPI, instead of worrying about the numerous infrastructure details the users simply enrich their function's business logic with the function annotations to instruct the self-provisioning infrastructure with the additional information that is required to provision the complete function's execution environment.

Self-Provisioning Storage Attachments

One of the main design decisions of serverless computing is the complete desegregation of compute and storage [69]. Such disaggregation offers benefits such as flexible scaling of compute resources, but at the same time, it requires the user to provision storage attachments for the non-trivial

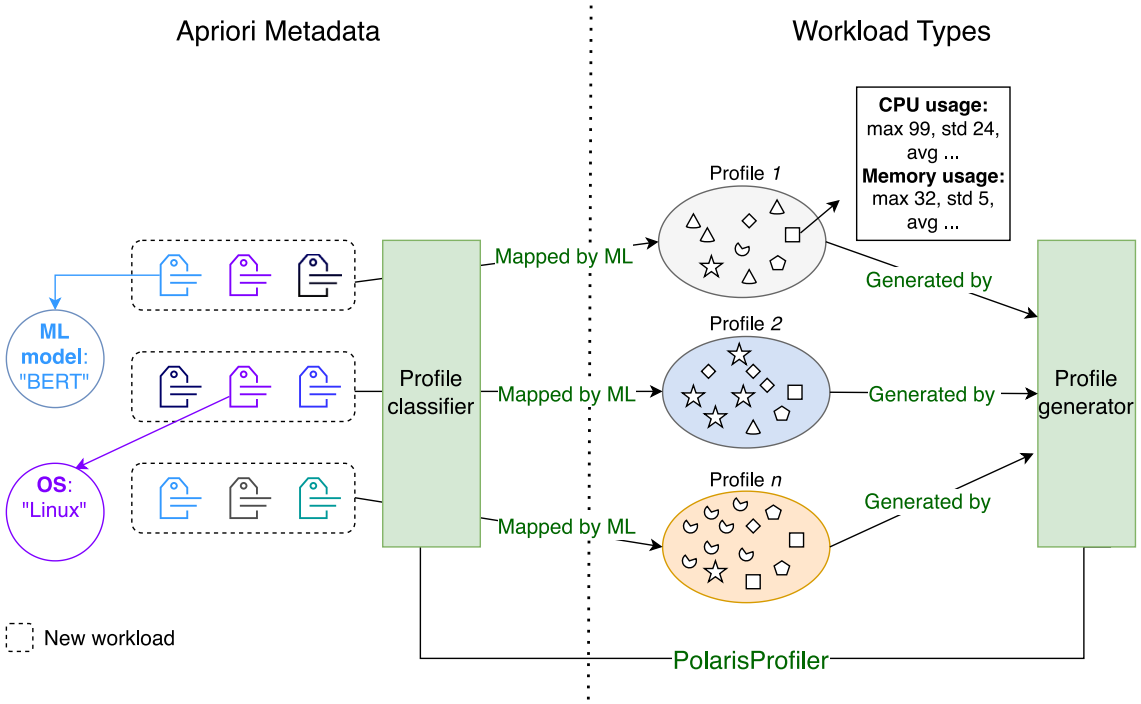

Fig. 3 Overview of the metadata-based profiling

serverless applications [8]. Serverless applications typically use object stores to persist their results [70], but also to transfer the intermediary, ephemeral data between the steps in serverless workflows. In practice, this means that the users typically have to provision and manage a large number of object store buckets (together with their access policies and configuration model), many of which have only a very limited lifespan. This makes the provisioning and management of the storage attachments the "low-hanging fruit" that needs to be handled by the next generation of serverless computing.

The self-provisioning infrastructures address this challenge by offering the self-provisioning of serverless storage attachments. In this example, we focus on the object stores, but a similar approach can be applied to other storage types such as key-value stores [71]. To provision an adequate storage attachment the infrastructure first needs to know that a specific function needs to write to or read from a storage attachment. This is relatively straightforward to obtain from the application's business logic. For example, this can be done based on the used primitives or even proprietary SDK calls such as *S3.putObject()*. Please note that the special case where the function is triggered by a change in a storage attachment is discussed in the previous example on *Self-provisioning FaaS triggers*. Next, the infrastructure needs to determine how to "wire" the access to the storage attachments, because multiple functions can share the same storage attachment, e.g., in the case of passing intermediary data between two functions in a serverless workflow. The shared storage attachments can be identified by looking at the function triggers, direct function invocations, and internal state mapping. The internal state mapping stores all the provisioned resources and configuration is maintained by the SPI. The SPI can use this internal representation of the infrastructure state to infer the bucket (or table) references in situations such as serverless workflow, where there are no explicit connections between the functions. At this stage, it is trivial to extrapolate access policies with suitable access rights. Regarding the storage attachment configuration models, they can be self-provisioned following an approach similar to the one presented next.

Self-Provisioning of Function Resources and Configurations

The contemporary serverless computing paradigm says nothing about the management of the configuration model. This is true not only for the BaaS services but also for the configurations of the FaaS functions. Most of the FaaS platforms provide elaborate configuration options. For example, besides the well-known memory/CPU configuration, one can also usually specify maximum and minimum container instances (containers warm pool), snap start - cached container image, maximal concurrent requests per container instance, ephemeral storage size (e.g., "/tm" directory), and so forth. This is not only complex for the users, but due to the current approach to the serverless configuration

options, we often end up with sub-optimal configurations for the serverless functions. This can lead to, for example, significantly over-provisioned memory [26].

The SPI addresses these challenges, by providing self-optimized configuration models and resource allocations. For example, to deploy a function one needs to specify its configuration model. This at least requires configuring the function's memory/CPU resources. Even for this simple case, the users have to conduct custom experiments to determine the right configuration, rely on their intuition, or decide to over-provision "just in case". The SPI can easily self-optimize the memory/CPU configuration in a two-stage approach. Firstly, the SPI uses Metadata-based profiling to determine the function's profile. More specifically, it looks at the function's metadata, which is known also before function deployment. Based on the metadata it selects the most suitable profile that contains the optimal memory/CPU configuration. After this, the function can already be self-provisioned by the SPI. Secondly, the SPI can rely on a range of optimizations to perform further configuration tuning if necessary. Examples include statistical learning methods such as Gaussian Process Bayesian Optimization [27, 48], analytical models such as Probability Refined Critical Path Greedy algorithm [45] or Tree-structured Parzen Estimator [28], and AI-based approaches such as Deep Neural Networks [72] and Reinforcement Learning [73–75].

By following a similar approach the SPI can also self-optimize resource and configuration models for the BaaS services.

Self-Updating Mechanism

Next, we analyze mechanisms for self-provisioning of non-functional aspects. Naturally, there are many non-functional concerns and we cannot address them all in a single paper. Instead, we focus on a well-known concern and mechanisms for performing software updates.

One of the advantages of the current serverless computing paradigm and of using managed BaaS is that the application developers do not have to worry about the software updates both of the managed BaaS services and the application runtime. For example, a serverless database or a key-value store is typically delivered as managed services that are continuously updated (and patched) to their latest version by the platform provider. However, at the moment if a function needs to utilize or simply can benefit from a new feature or patch that was delivered with the latest update, the function's dependencies need to be updated, the function's source code potentially needs to be changed and a new version of the function needs to be rolled out, as well. Contrary to this, the self-provisioning infrastructure can utilize its knowledge of the application's source code and the registered *Infrastructure capabilities* to provide a suitable implementation of the *Serverless primitives*, which is able to utilize the latest infrastructure features seamlessly, without changing the function's source code and having to roll out a new version. More concretely, since the serverless function only depends on the Serverless primitives and not on the specific Infrastructure capabilities of the serverless infrastructure nor on the concrete Capability Providers such as Redis for the key-value store (cf. Fig. 1), the SPI can transparently provide a new Capability implementation and switch the function from an old version of the Capability to a new one without the function even noticing it. With this technique, the SPI enables delivering true self-upgrading capabilities. Similarly, we can also transparently change the capability providers or even core infrastructure components, but this is out of the scope of this paper.

Discussion

The SPIs introduce abstractions and techniques to enable enhanced portability, interoperability, and utility-based consumption of the infrastructure resources. These elements are typically absent in contemporary serverless models. We also showed that by leveraging AI-driven metadata-based profiling, self-provisioning infrastructure autonomously refines infrastructure configuration and resource allocation, ensuring optimized performance, efficiency, and cost-effectiveness. The SPIs introduce runtime mixins to introduce SLO awareness [76] to the self-provisioning infrastructure and set foundations toward achieving further non-functional properties such as self-reliability, self-auditing, and self-governance. Next, we discuss some of the trade-offs and the practical applicability of the self-provisioning infrastructures.

Learning curve The self-provisioning infrastructure requires the usage of custom programming abstractions, so-called serverless primitives. Despite being well-aligned with existing serverless programming models, they still put an additional burden on the developers, because the developers need to learn how to work with the new API and interfaces. This can be seen as a limiting factor in terms of its practical applicability. Fortunately, as a consequence of their design, the self-provisioning infrastructure can be gradually adopted. It is possible to adopt only a subset of the proposed models and still receive benefits from the self-provisioning infrastructure paradigm. Such gradual adaption can lead to the flattening of the learning curve, hence mitigating most of the drawbacks that accompany the introduction of new abstractions. Introducing the serverless primitives as a novel abstraction was a deliberate design decision since we believe that the benefits they bring outweigh their potential drawbacks.

Adoption of infrastructure capabilities Some of the core enablers of the self-provisioning infrastructures are the

so-called infrastructure capabilities. The proposed capabilities need to be created and maintained manually, including their provisioning logic. This can be a time-consuming task, especially considering that the underlying BaaS services can add new features (frequent), change their APIs (infrequent), and add new provisioning options (relatively frequent). Additionally, there needs to be a certain critical mass of the implemented infrastructure capabilities before the self-provisioning infrastructure can transition from the research prototype to being production-ready. All these drawbacks can lead to a reduced practical application of the SPIs. However, there are several options to mitigate most of the mentioned drawbacks. Firstly, as previously mentioned, the self-provisioning infrastructures can be adopted gradually. Consequently, the same is true for their rollout. This means that not all the functionality has to be provided and delivered at once, reducing the burden on the capability providers. Secondly, the self-provisioning infrastructure can utilize the power of the open-source community to deliver and maintain the infrastructure capabilities. The large cloud providers are particularly incentivized to include their capability providers to ensure their services can be consumed by the next-generation serverless application. Finally, large language models (LLMs) are poised to play a more prominent role in future code generation, hence potentially leading to solutions for auto-generating the infrastructure capabilities.

Portability and interoperability The serverless primitives, together with the infrastructure capabilities are designed based on the least common denominator of the cloud BaaS services. This leads to a drawback since the default primitives and capabilities can fail to capture all the features available in various implementations of BaaS services. As described in "Principles and Models of Self-Provisioning Infrastructure" section, this can be partly mitigated by providing a custom implementation of the primitives and capabilities. However, as a consequence, there is a decrease in portability and interoperability that also leads to a vendor lock-in. Therefore, designing the capabilities based on the principle of the least common denominator was a deliberate design decision because the self-provisioning infrastructure aims to increase interoperability and enable next-generation serverless computing to support federated FaaS, sky computing, and edge-cloud deployments.

Conclusion

In this paper, we introduced self-provisioning infrastructure as a novel paradigm that enables next-generation serverless computing. We showed how the self-provisioning infrastructure fortifies the core principles of serverless computing and extends the serverless paradigm holistically across the entire application execution environment and infrastructure. We presented and analyzed the design principles, models, and mechanisms of the self-provisioning infrastructure.

In the future, we intend to continue the research in the self-provisioning infrastructures. In particular, we will aim to extend the self-provisioning infrastructure in several directions. Firstly, we intend to focus on applications' lifecycle beyond the provisioning phase to support the next-generation serverless applications during their runtime. Specifically, we intend to address the performance, reliability, and AI/edge-intelligence-specific challenges in the Edge-Cloud continuum. Secondly, we plan to extend the self-provisioning infrastructure to facilitate a paradigm shift from traditional services and platforms computing to fabric-centric computing where digital resources, infrastructures, and systems become true utilities, that permeate the entire computational and data continuum [12]. Finally, we plan to extend the self-provisioning infrastructure's runtime mechanisms to offer structured support for self-auditing, self-compliance, and self-governance.

Acknowledgements This work is sponsored by the Austrian Research Promotion Agency (FFG), under project No. 903884.

Author Contributions The author confirms sole responsibility for conception and design, material preparation, data collection and analysis.

Funding Open access funding provided by TU Wien (TUW). This work is sponsored by the Austrian Research Promotion Agency (FFG), under project No. 903884.

Data Availability Not applicable.

Declarations

Conflict of interest The author has no Conflict of interest to declare that are relevant to the content of this article.

Informed Consent Not applicable.

Research Involving Human and /or Animals No/Not applicable.

References

1. Jonas E, Schleier-Smith J, Sreekanti V, Tsai C, Khandelwal A, Pu Q, Shankar V, Carreira J, Krauth K, Yadwadkar NJ, Gonzalez JE, Popa RA, Stoica I, Patterson DA. Cloud programming simplified: a Berkeley view on serverless computing. CoRR; 2019. arXiv:1902.03383
2. Nastic S, Rausch T, Scekic O, Dustdar S, Gusev M, Koteska B, Kostoska M, Jakimovski B, Ristov S, Prodan R. A serverless real-time data analytics platform for edge computing. IEEE Internet Comput. 2017;21(4):64–71. https://doi.org/10.1109/MIC.2017.2911430.
3. Raith P, Nastic S, Dustdar S. Serverless edge computing-where we are and what lies ahead. IEEE Internet Comput. 2023;27(3):50–64. https://doi.org/10.1109/MIC.2023.3260939.
4. Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P. In: Chaudhary S, Somani G, Buyya R, editors. Serverless computing: current trends and open problems. Singapore: Springer; 2017. p. 1–20. https://doi.org/10.1007/978-981-10-5026-8_1.
5. Du D, Liu Q, Jiang X, Xia Y, Zang B, Chen H. Serverless computing on heterogeneous computers. In: ASPLOS '22. Association for Computing Machinery, New York, NY, USA; 2022. p. 797–813. https://doi.org/10.1145/3503222.3507732
6. Jia Z, Witchel E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: ASPLOS '21. Association for Computing Machinery, New York, NY, USA; 2021. p. 152–66. https://doi.org/10.1145/3445814.3446701
7. Castro P, Ishakian V, Muthusamy V, Slominski A. The rise of serverless computing. Commun ACM. 2019;62(12):44–54. https://doi.org/10.1145/3368454.
8. Eismann S, Scheuner J, Eyk E, Schwinger M, Grohmann J, Herbst N, Abad C, Iosup A. Serverless applications: why, when, and how? IEEE Softw. 2021;38(1):32–9. https://doi.org/10.1109/MS.2020.3023302.
9. Li Z, Guo L, Cheng J, Chen Q, He B, Guo M. The serverless computing survey: a technical primer for design architecture. ACM Comput Surv. 2022;54(10s):1–34. https://doi.org/10.1145/3508360.
10. Safaryan G, Jindal A, Chadha M, Gerndt M. Slam: Slo-aware memory optimization for serverless applications. In: 2022 IEEE 15th international conference on cloud computing (CLOUD); 2022. p. 30–9. https://doi.org/10.1109/CLOUD55607.2022.00019
11. Milojicic D. The edge-to-cloud continuum. Computer. 2020;53(11):16–25. https://doi.org/10.1109/MC.2020.3007297.
12. Nastic S, Dustdar S, Philipp R, Alireza F, Pusztai T. A serverless computing fabric for edge & cloud. In: 4th IEEE international conference on cognitive machine intelligence (CogMi); 2022. https://doi.org/10.1109/CogMI56440.2022.00011
13. Dustdar S, Pujol VC, Donta PK. On distributed computing continuum systems. IEEE Trans Knowl Data Eng. 2023;35(4):4092–105. https://doi.org/10.1109/TKDE.2022.3142856.
14. Babuji Y, Bryan J, Chard R, Chard K, Foster I, Galewsky B, Katz DS, Li Z. Federated function as a service for escience. In: 2021 IEEE 17th international conference on eScience (eScience). IEEE Computer Society, Los Alamitos, CA, USA; 2021. p. 251–2. https://doi.org/10.1109/eScience51609.2021.00046
15. Chard R, Babuji Y, Li Z, Skluzacek T, Woodard A, Blaiszik B, Foster I, Chard K. Funcx: A federated function serving fabric for science. In: Proceedings of the 29th international symposium on high-performance parallel and distributed computing. HPDC '20. Association for Computing Machinery, New York, NY, USA; 2020. p. 65–76. https://doi.org/10.1145/3369583.3392683
16. Li Z, Chard R, Babuji Y, Galewsky B, Skluzacek TJ, Nagaitsev K, Woodard A, Blaiszik B, Bryan J, Katz DS, Foster I, Chard K. funcx: federated function as a service for science. IEEE Trans Parallel Distrib Syst. 2022;33(12):4948–63. https://doi.org/10.1109/TPDS.2022.3208767.
17. Ristov S, Gritsch P. FaaSt: optimize makespan of serverless workflows in federated commercial FaaS. In: 2022 IEEE international conference on cluster computing. CLUSTER '22. IEEE, Heidelberg, Germany; 2022. p. 182–94. https://doi.org/10.1109/CLUSTER51413.2022.00032
18. Ristov S, Pedratscher S, Fahringer T. xAFCL: run scalable function choreographies across multiple FaaS systems. IEEE Trans Serv Comput. 2023;16(1):711–23. https://doi.org/10.1109/TSC.2021.3128137.
19. Sampe J, Garcia-Lopez P, Sanchez-Artigas M, Vernik G, Roca-Llaberia P, Arjona A. Toward multicloud access transparency in serverless computing. IEEE Softw. 2021;38(1):68–74. https://doi.org/10.1109/MS.2020.3029994.
20. Stoica I, Shenker S. From cloud computing to sky computing. In: Proceedings of the workshop on hot topics in operating systems. HotOS '21. Association for Computing Machinery, New York, NY, USA; 2021. p. 26–32. https://doi.org/10.1145/3458336.3465301
21. Castro P, Isahagian V, Muthusamy V, Slominski A. In: Krishnamurthi R, Kumar A, Gill SS, Buyya R, editors. Hybrid serverless computing: opportunities and challenges. Cham: Springer; 2023. p. 43–77. https://doi.org/10.1007/978-3-031-26633-1_3.
22. Durillo JJ, Prodan R, Barbosa JG. Pareto tradeoff scheduling of workflows on federated commercial clouds. Simul Model Pract Theory. 2015;58:95–111. https://doi.org/10.1016/j.simpat.2015.07.001.
23. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B. Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 symposium on cloud computing. SoCC '17. Association for Computing Machinery, New York, NY, USA; 2017. p. 445–51. https://doi.org/10.1145/3127479.3128601
24. Sampe J, Sanchez-Artigas M, Vernik G, Yehekzel I, Garcia-Lopez P. Outsourcing data processing jobs with lithops. IEEE Trans Cloud Comput. 2021. https://doi.org/10.1109/TCC.2021.3129000.
25. Pedratscher S, Ristov S, Fahringer T. M2FaaS: transparent and fault tolerant FaaSification of node.js monolith code blocks. Fut Gener Comput Syst. 2022;135:57–71. https://doi.org/10.1016/j.future.2022.04.021.
26. Tian H, Li S, Wang A, Wang W, Wu T, Yang H. Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In: Proceedings of the 13th symposium on cloud computing. SoCC '22. Association for Computing Machinery, New York, NY, USA. p. 78–93. https://doi.org/10.1145/3542929.3563470
27. Akhtar N, Raza A, Ishakian V, Matta I. Cose: configuring serverless functions using statistical learning. In: IEEE INFOCOM 2020—IEEE conference on computer communications; 2020. p. 129–38. https://doi.org/10.1109/INFOCOM41043.2020.9155363
28. Yu G, Chen P, Zheng Z, Zhang J, Li X, He Z. FaaSdeliver: cost-efficient and QoS-aware function delivery in computing continuum. IEEE Trans Serv Comput. 2023;16(5):3332–47. https://doi.org/10.1109/TSC.2023.3274769.
29. HasiCorp: Terraform: automate infrastructure on any cloud. https://github.com/hashicorp/terraform. Accessed 13 Oct 2023
30. Services AW. CloudFormation: speed up cloud provisioning with infrastructure as code. https://github.com/aws-cloudformation. Accessed 13 Oct 2023
31. Wurster M, Breitenbücher U, Képes K, Leymann F, Yussupov V. Modeling and automated deployment of serverless applications using TOSCA. In: 2018 IEEE 11th conference on service-oriented

computing and applications (SOCA); 2018. p. 73–80. https://doi.org/10.1109/SOCA.2018.00017
32. Inc., S.: Serverless: build web, mobile and IoT applications with serverless architectures. https://github.com/serverless/serverless. Accessed 13 Oct 2023
33. Hat R. Ansible: ansible is a radically simple IT automation system. https://github.com/ansible/ansible. Accessed 13 Oct 2023
34. Services AW. AWS CDK: a framework for defining cloud infrastructure in code. https://github.com/aws/aws-cdk. Accessed 13 Oct 2023
35. Corporation P. Pulumi—infrastructure as code in any programming language. https://github.com/pulumi/pulumi. Accessed: 2023-10-13
36. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B. Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 symposium on cloud computing. SoCC '17. Association for Computing Machinery, Santa Clara, CA; 2017. p. 445–51. https://doi.org/10.1145/3127479.3128601
37. Cordingly R, Shu W, Lloyd WJ. Predicting performance and cost of serverless computing functions with SAAF. In: IEEE DASC/PiCom/CBDCom/CyberSciTech; 2020. p. 640–9. https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech49142.2020.00111
38. Chard R, Babuji Y, Li Z, Skluzacek T, Woodard A, Blaiszik B, Foster I, Chard K. FuncX: a federated function serving fabric for science. In: International symposium on high-performance on parallel and distributed computing. HPDC '20. ACM, Stockholm, Sweden; 2020. p. 65–76. https://doi.org/10.1145/3369583.3392683
39. Tankov V, Valchuk D, Golubev Y, Bryksin T. Infrastructure in code: towards developer-friendly cloud applications. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE); 2021. p. 1166–70. https://doi.org/10.1109/ASE51524.2021.9678943
40. Services AW. Aws Chalice: python serverless microframework for AWS. https://github.com/aws/chalice/. Accessed 13 Oct 2023
41. Developers Z. Zappa: serverless python. https://github.com/zappa/Zappa/. Accessed 13 Oct 2023
42. Developers O. Osiris: simple serverless web applications in Kotlin. https://github.com/cjkent/osiris/. Accessed 13 Oct 2023
43. Yussupov V, Breitenbücher U, Kaplan A, Leymann F. Seaport: assessing the portability of serverless applications. In: CLOSER; 2020. p. 456–67
44. Elgamal T, Sandur A, Nahrstedt K, Agha G. Costless: optimizing cost of serverless computing through function fusion and placement. In: 2018 IEEE/ACM symposium on edge computing (SEC); 2018. p. 300–12. https://doi.org/10.1109/SEC.2018.00029
45. Lin C, Khazaei H. Modeling and optimization of performance and cost of serverless applications. IEEE Trans Parallel Distrib Syst. 2021;32(3):615–32. https://doi.org/10.1109/TPDS.2020.3028841.
46. Wen Z, Wang Y, Liu F. Stepconf: Slo-aware dynamic resource configuration for serverless function workflows. In: IEEE INFOCOM 2022—IEEE conference on computer communications; 2022. p. 1868–77. https://doi.org/10.1109/INFOCOM48880.2022.9796962
47. Zubko T, Jindal A, Chadha M, Gerndt M. Maff: self-adaptive memory optimization for serverless functions. In: Montesi F, Papadopoulos GA, Zimmermann W, editors. Service-oriented and cloud computing. Cham: Springer International Publishing; 2022. p. 137–54.
48. Alipourfard O, Liu HH, Chen J, Venkataraman S, Yu M, Zhang M. Cherrypick: adaptively unearthing the best cloud configurations for big data analytics. In: 14th USENIX symposium on networked systems design and implementation (NSDI 17). USENIX Association, Boston, MA; 2017. p. 469–82. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard
49. Cordingly R, Xu S, Lloyd W. Function memory optimization for heterogeneous serverless platforms with CPU time accounting. In: 2022 IEEE international conference on cloud engineering (IC2E); 2022. p. 104–15. https://doi.org/10.1109/IC2E55432.2022.00019
50. Eismann S, Bui L, Grohmann J, Abad C, Herbst N, Kounev S. Sizeless: predicting the optimal size of serverless functions. In: Proceedings of the 22nd international middleware conference. Middleware '21. Association for Computing Machinery, New York, NY, USA; 2021. p. 248–59. https://doi.org/10.1145/3464298.3493398
51. Amazon Web Services, Inc. AWS Compute Optimizer; 2023. https://aws.amazon.com/compute-optimizer/. Accessed 13 Oct 2023
52. Raza A, Akhtar N, Isahagian V, Matta I, Huang L. Configuration and placement of serverless applications using statistical learning. IEEE Trans Netw Serv Manag. 2023;20(2):1065–77. https://doi.org/10.1109/TNSM.2023.3254437.
53. Yu G, Chen P, Zheng Z, Zhang J, Li X, He Z. FaaSdeliver: cost-efficient and QoS-aware function delivery in computing continuum. IEEE Trans Serv Comput. 2023. https://doi.org/10.1109/TSC.2023.3274769.
54. Finnie-Ansley J, Denny P, Becker BA, Luxton-Reilly A, Prather J. The robots are coming: exploring the implications of openAI codex on introductory programming. In: Proceedings of the 24th Australasian computing education conference. ACE '22. Association for Computing Machinery, New York, NY, USA; 2022. p. 10–9. https://doi.org/10.1145/3511861.3511863
55. Pearce H, Tan B, Ahmad B, Karri R, Dolan-Gavitt B. Can openAI codex and other large language models help us fix security bugs? CoRR; 2021. arXiv:2112.02125.
56. Prenner JA, Robbes R. Automatic program repair with openAI's codex: evaluating quixbugs. CoRR; 2021. arXiv:2111.03922
57. Thurner P. Seamless AI-powered observability for multicloud serverless applications. Dynatrace News. https://www.dynatrace.com/news/blog/seamless-ai-powered-observability-for-serverless/
58. Masood A, Hashmi A. AIOps: predictive analytics & machine learning in operations. Berkeley, CA: Apress; 2019. p. 359–82. https://doi.org/10.1007/978-1-4842-4106-6_7.
59. Levin A, Garion S, Kolodner EK, Lorenz DH, Barabash K, Kugler M, McShane N. Aiops for a cloud object storage service. In: 2019 IEEE international congress on big data (BigDataCongress); 2019. p. 165–9. https://doi.org/10.1109/BigDataCongress.2019.00036
60. Thurner P. Artificial intelligence for IT operations (AIOps). https://www.ibm.com/cloud/learn/aiops. Accessed 13 Oct 2023
61. Glikson A, Nastic S, Dustdar S. Deviceless edge computing: extending serverless computing to the edge of the network. In: Proceedings of the 10th ACM international systems and storage conference. SYSTOR '17. Association for Computing Machinery, New York, NY, USA; 2017. https://doi.org/10.1145/3078468.3078497
62. Developers WCC. A generic way for WASI applications to interact with cloud services. https://github.com/WebAssembly/wasi-cloud-core
63. Developers O. Libcloud. https://libcloud.apache.org/. Accessed 13 Oct 2023
64. Bracha G, Cook W. Mixin-based inheritance. In: Proceedings of the European conference on object-oriented programming on object-oriented programming systems, languages, and applications. OOPSLA/ECOOP '90. Association for Computing Machinery, New York, NY, USA; 1990. p. 303–11. https://doi.org/10.1145/97945.97982
65. Guo H, Yang J, Liu J, Yang L, Chai L, Bai J, Peng J, Hu X, Chen C, Zhang D, Shi X, Zheng T, Zheng L, Zhang B, Xu K, Li

Z. OWL: a large language model for IT operations. arXiv:2309.09298. https://doi.org/10.48550/arXiv.2309.09298
66. Morichetta A, Casamayor Pujol V, Nastic S, Dustdar S, Vij D, Xiong Y, Zhang Z. Polarisprofiler: a novel metadata-based profiling approach for optimizing resource management in the edge-cloud continnum. In: The 18th IEEE international symposium on service-oriented system engineering (SOSE 2023); 2023. https://doi.org/10.1109/SOSE58276.2023.00010
67. Li K, Nastic S. Attentionfunc: balancing FaaS compute across edge-cloud continuum with reinforcement learning. In: The 13th international conference on the internet of things (IoT 2023); 2023. (to appear)
68. Losio R. Are recursive serverless functions the biggest billing risk on the cloud? https://www.infoq.com/news/2022/08/recursive-serverless-functions/
69. Mvondo D, Bacou M, Nguetchouang K, Ngale L, Pouget S, Kouam J, Lachaize R, Hwang J, Wood T, Hagimont D, De Palma N, Batchakui B, Tchana A. Ofc: an opportunistic caching system for FaaS platforms. In: Proceedings of the sixteenth European conference on computer systems. EuroSys '21. Association for Computing Machinery, New York, NY, USA; 2021. p. 228–44. https://doi.org/10.1145/3447786.3456239
70. Sánchez-Artigas M, Eizaguirre GT. A seer knows best: optimized object storage shuffling for serverless analytics. In: Proceedings of the 23rd ACM/IFIP international middleware conference. Middleware '22. Association for Computing Machinery, New York, NY, USA; 2022. p. 148–60. https://doi.org/10.1145/3528535.3565241
71. Klimovic A, Wang Y, Stuedi P, Trivedi A, Pfefferle J, Kozyrakis C. Pocket: elastic ephemeral storage for serverless analytics. In: 13th USENIX symposium on operating systems design and implementation (OSDI 18). USENIX Association, Carlsbad, CA; 2018. p. 427–44. https://www.usenix.org/conference/osdi18/presentation/klimovic
72. Jindal A, Chadha M, Benedict S, Gerndt M. Estimating the capacities of function-as-a-service functions. In: Proceedings of the 14th IEEE/ACM international conference on utility and cloud computing companion. UCC '21. Association for Computing Machinery, New York, NY, USA; 2022. https://doi.org/10.1145/3492323.3495628
73. Bitsakos C, Konstantinou I, Koziris N. Derp: a deep reinforcement learning cloud system for elastic resource provisioning. In: 2018 IEEE international conference on cloud computing technology and science (CloudCom); 2018. p. 21–9. https://doi.org/10.1109/CloudCom2018.2018.00020
74. Schuler L, Jamil S, Kuhl N. AI-based resource allocation: reinforcement learning for adaptive auto-scaling in serverless environments. In: 2021 IEEE/ACM 21st international symposium on cluster, cloud and internet computing (CCGrid). IEEE Computer Society, Los Alamitos, CA, USA; 2021. p. 804–11. https://doi.org/10.1109/CCGrid51090.2021.00098
75. Benedetti P, Femminella M, Reali G, Steenhaut K. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In: 2022 IEEE international conference on pervasive computing and communications workshops and other affiliated events (PerCom Workshops); 2022. p. 674–9. https://doi.org/10.1109/PerComWorkshops53856.2022.9767437
76. Nastic S, Morichetta A, Pusztai T, Dustdar S, Ding X, Vij D, Xiong Y. Sloc: service level objectives for next generation cloud computing. IEEE Internet Comput. 2020;24(3):39–50. https://doi.org/10.1109/MIC.2020.2987739.