SLATE: Managing Heterogeneous Cloud Functions

Jessica Vandebon*, José G. F. Coutinho*, Wayne Luk*, Eriko Nurvitadhi† and Mishali Naik†
*Imperial College London, United Kingdom
{jessica.vandebon17, gabriel.figueiredo, w.luk}@imperial.ac.uk
†Intel Corporation, San Jose, USA
{eriko.nurvitadhi, mishali.naik}@intel.com

***Abstract*—This paper presents SLATE, a fully-managed, heterogeneous Function-as-a-Service (FaaS) system for deploying serverless functions onto heterogeneous cloud infrastructures. We extend the traditional homogeneous FaaS execution model to support *heterogeneous functions*, automating and abstracting runtime management of heterogeneous compute resources in order to improve cloud tenant accessibility to specialised, accelerator resources, such as FPGAs and GPUs. In particular, we focus on the mechanisms required for *heterogeneous scaling* of deployed function instances to guarantee latency objectives while minimising cost. We develop a simulator to validate and evaluate our approach, considering case-study functions in three application domains: machine learning, bio-informatics, and physics. We incorporate empirically derived performance models for each function implementation targeting a hardware platform with combined computational capacity of 24 FPGAs and 12 CPU cores. Compared to homogeneous CPU and homogeneous FPGA functions, simulation results achieve respectively a cost improvement for non-uniform task traffic of up to 8.7 times and 1.7 times, while maintaining specified latency objectives.**

I. Introduction

In the last decade, cloud computing has shaped the information technology landscape, with increasing numbers of businesses and researchers offloading their computation needs to cloud data centres to drastically reduce their operating costs. Cloud IaaS (Infrastructure-as-a-Service) systems, such as Amazon EC2 [1], provide on-demand virtual resources, such as servers, routers and storage. Since physical resources can be shared across different virtual instances, clients are able to save costs and providers can maximise resource utilisation. With IaaS, however, cloud tenants may still need to load-balance their computing resources, and to dynamically readjust allocated resources according to demand in order to save costs. In contrast, PaaS (Platform-as-a-Service) systems automate these efforts, managing provisioned resources subject to allocation and scaling rules provided by tenants.

Recently, a new cloud model has emerged called **FaaS** (Function-As-A-Service), which simplifies pricing, management, and deployment over both PaaS and IaaS. FaaS is event-driven, centred around requests for code execution (functions). This code runs inside stateless containers, and can be triggered by different event types such as web requests, database events, queuing services, and monitoring alerts. With FaaS, clients pay for each serviced request (Fig. 1(b)), offloading resource allocation and balancing responsibilities to the provider. This pricing model differs from IaaS and PaaS models, in which tenants pay for allocated cloud resources for the period in which they have them, regardless of whether they are used or are idle (Fig. 1(a)). FaaS has found its place in major cloud platforms (e.g. AWS Lambda [2], Microsoft Azure Functions [3], and Google Cloud Functions [4]), supporting real-time data processing (batch and stream processing), Internet of things (IoT), and edge computing.

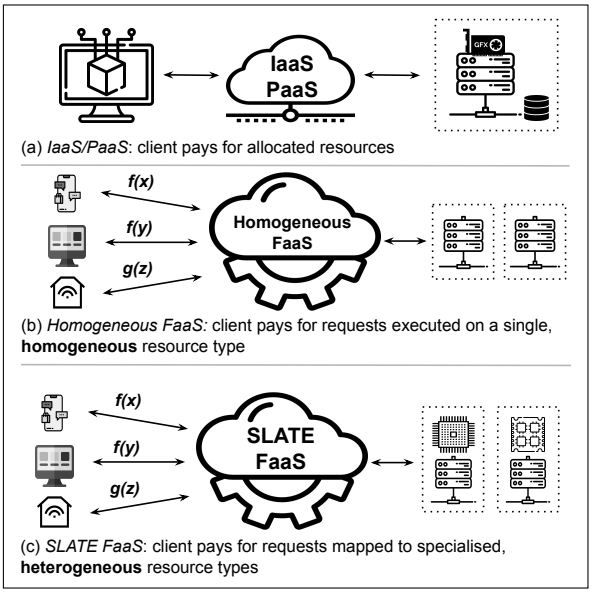

Fig. 1. Comparison between **(a)** resource-oriented cloud models, **(b)** homogeneous FaaS, and **(c)** heterogeneous **SLATE** FaaS.

In this paper, we present **SLATE** (Heterogeneou**S** c**L**oud m**A**nagement for Func**T**ion-as-a-service Syst**E**ms), a novel heterogeneous FaaS approach (Fig. 1(c)) designed to leverage heterogeneous cloud compute resources, such as CPUs and FPGAs, in order to provide further performance and cost benefits over traditional homogeneous FaaS approaches. **SLATE** is designed for scenarios where functions have very different computational requirements and latency (timing) objectives. In particular, current FaaS offerings are limited to horizontal scaling: users identify a single resource configuration to serve any given request, and the system spawns replicas of the same configuration unit according to demand. While this approach is well-suited for web applications which scale uniformly with one type of resource, it does not address performance and pricing concerns when considering requests with different computational requirements in domains such as High Performance Computing (HPC) and Artificial Intelligence.

The main contributions of this paper are as follows:

1) The **SLATE** FaaS architecture and management mechanisms (Section II);
2) The implementation of a simulated FaaS prototype with the above architecture (Section III-A);
3) An evaluation of our prototype targeting three application domains, namely machine learning, bioinformatics, and physics, on FPGA and CPU resources. We compare our heterogeneous FaaS system to current FaaS systems in terms of performance (Section III-C) and cost (Section III-D).

II. Approach

A. Overview

Let us consider a scenario where an application employs two cloud functions that perform Machine Learning (ML) tasks, namely: *training* and *inference*. Model training requires sending large chunks of data at regular time intervals, while inference tasks are smaller and happen irregularly according to user demand. In this example, we have two distinct task types with specific performance requirements: training tasks process bulk data and are more computationally intensive, while inference tasks are smaller and have lighter computation requirements.

Current FaaS solutions are not designed to support such scenarios, in which tasks have very different computation requirements. In particular, users must identify a single resource configuration (e.g. a 4 core CPU with 512MB of RAM) to service every incoming request. Every time a request is submitted, the FaaS platform uses a replica of the same resource configuration instance to execute that task, and clients pay per request serviced. So, in the case where we have heterogeneous traffic with both small (low computationally-intensive) and large (high computationally-intensive) tasks, the following applies with current FaaS solutions:

a) clients may ensure they have a large enough configuration to service both types of tasks, however this leads to over-provisioning and thus over-paying for smaller tasks;
b) if a resource configuration is heterogeneous (for example, includes both a CPU and an FPGA), clients need to manually load-balance traffic to distribute task workloads to the appropriate resource, for instance, sending smaller tasks to the CPU and larger tasks to the FPGA;
c) clients may try to identify the cheapest resource configuration that meets latency requirements for each type of task, however this requires expertise.

In general, when considering heterogeneous computations, there is no single resource configuration that works best for all types of workloads, and determining the best configuration for each scenario is not obvious. For instance, smaller jobs may perform faster on CPUs since data movement and offload overheads would dominate otherwise, while sufficiently large streaming and data-parallel workloads may perform better on FPGAs and GPUs, respectively. Moreover, data-types and numerical representations may also drastically affect relative performance. For instance, FPGAs tend to excel with integer-based operations, while CPUs and GPUs are designed to work with double-precision operations. Thus, management techniques based solely on horizontal scaling do poorly to leverage the benefits of heterogeneous computation.

The lack of support for heterogeneity in cloud computing in general, and FaaS in particular, can be attributed to the complexity of its runtime management. In addition to balancing task requests horizontally across devices to service as many requests as possible in parallel, it becomes necessary to scale requests vertically according to the device type that is best suited to service it. With new accelerators appearing in the market every year, management logic needs to be flexible and generic to support legacy and new devices. Knowledge about the suitability of each resource to different workloads is necessary, but acquiring and maintaining such knowledge is challenging, particularly as platforms grow.

We have designed **SLATE**, an FaaS approach that supports heterogeneous tasks and resources, and addresses the above scenarios. First, clients need to specify the functions that they wish to execute, and how fast each function needs to run (latency requirement). Clients can optionally restrict the domain for each function, which may reduce the number of candidate resource configurations and reduce pricing. Our FaaS system will then automatically identify the cheapest resource configurations that can meet the timing constraints for each function. Note that function domains are automatically segmented to find the most appropriate resource configuration for each sub-domain. When submitting a function request, our system will automatically map that task to the most appropriate resource configuration, spawning a new instance if there is none currently available. In this way, we meet timing constraints in a cost efficient manner for every task executed. The following sections outline the **SLATE** design and the mechanisms that enable our heterogeneous FaaS approach.

B. Execution

In this section, we focus on the key components of the **SLATE** architecture and its execution.

We begin with the definitions used throughout the remainder of this paper. A cloud *function* is a computation available for execution by the FaaS system. A function *request* defines a task that we we wish to execute, for instance $matmul(A, B)$ where A and B are $N \times N$ matrices. Requests are resource-oblivious. Each request is serviced by a *function instance*, which is resource configuration allocated by the FaaS system to service that task. Each instance has an associated *function type*, (N, PE, f, D), where N and PE specify the resource configuration (quantity and type of processing element), f specifies the cloud function, and D specifies the supported input domain. Note that we can combine multiple processing elements ($N > 1$) into a single instance in order to acquire more computational power. Finally, each resource type is associated to a cost. The pricing is determined by the cloud provider and may dynamically change due to supply and demand considerations.

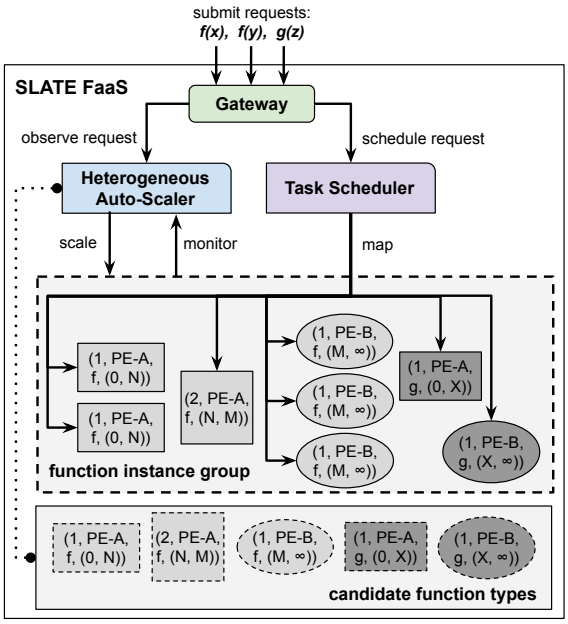

Fig. 2. Key components of the **SLATE** FaaS Architecture. **SLATE** allocates a function instance to service each request $f(x)$, where f is the cloud function and x is the input data.

Before execution, clients configure the **SLATE** system by submitting an application manifest. The manifest contains the list of all required functions, domains, latency (timing) requirements, and optional allocation rules. From the application manifest, **SLATE** determines the list of *candidate function types*, which are used during the life-cycle of the FaaS system. Once the system is configured, it is ready to service incoming requests and execute functions. We cover the configuration phase in more detail in II-C. For the remainder of this section we cover function execution.

The key components of the **SLATE** architecture are depicted in Fig. 2. The *gateway* serves as an interface for users and a single point of entry to the underlying FaaS resource management platform. To execute functions, users submit function requests to the *gateway*. The requests are forwarded to the *task scheduler* for execution, and to the *auto-scaler* for monitoring and scaling. The *instance group* contains all the allocated instances. Instances in the group are either idle and can be immediately employed by the task scheduler, or are busy executing a task. The *task scheduler* is responsible for mapping each request to a suitable instance in the group to meet latency constraints and minimise cost. The *auto-scaler*, on the other hand, is responsible for scaling the instance group to ensure the right resources are available, in quantity and type, for task execution. Next, we describe these processes in more detail:

- **Task Scheduler.** The task scheduler is responsible for mapping each request to an available, suitable instance in the group. An instance of type (N, PE, f, D) is suitable to execute a request $f(x)$ if $x \in D$. For example, a function instance with type $(2, GPU, matmul, (1000, 100000))$ can execute a matrix multiplication function using two GPUs, accepting $N \times N$ input matrices with $1000 \leq N \leq 100000$. During execution, the task scheduler selects a suitable instance from the group to execute the task, forwards the request to that instance for execution, and marks the instance as *busy* for the duration of execution.
- **Auto-Scaler.** The auto-scaler is responsible for adjusting the quantity and type of instances deployed in the instance group. It ensures that instances of suitable types are available to service incoming requests effectively. Auto-scaling is critical not only to maintain request throughput, but also to reduce the cloud provider's cost by minimising idle resources. The auto-scaler monitors each request and identifies suitable function types from the candidate list. If there are no available (i.e. not *busy*) suitable instances in the group, the auto-scaler spawns a new one.
  Furthermore, the auto-scaler maintains a log of the time and instance selected for every request. This log is checked periodically to determine each instance's *idle time*, i.e. the time since the last request for that instance type. If an instance's idle time is greater than the *idle time threshold* specified in the allocation rules (see Section II-C), and the instance is not currently busy, it is removed from the group. This contrasts to the auto-scaling mechanisms implemented in ORIAN [5] and other PaaS systems [6][7], which monitor incoming traffic and perform scaling events based on a window of past traffic (i.e. not at every request).

There are three important considerations in our system:

- **Zero-Scaling.** **SLATE** allows users to specify whether or not they want zero-scaling enabled. If zero-scaling is enabled, the system can scale an instance type down to zero replicas. If zero-scaling is disabled, **SLATE** ensures there is always one of each candidate instance type in the group to avoid start-up latency associated with spawning a new replica and initialising the function code.
- **Pricing.** The pricing models of current FaaS systems consist of two costs: (1) a *request cost*, which is a fixed rate per request, and (2) an *execution cost* which depends on the duration and resources used (e.g. memory and CPU) to execute a task. Unlike PaaS, tenants are not charged for allocated instances that are not used. With **SLATE**, the *request cost* is 0 when zero-scaling is enabled, and $\$(1\% \times min\_group\_cost)$ otherwise.
- **Performance Modelling.** Access to performance models is vital to make allocation decisions at runtime. In order to automate the modelling process, we use a generic offline method to collect samples for each function instance. The method is based on two assumptions: **(1)** throughput eventually saturates (stops changing) as we increase problem size, and **(2)** the implementation domain is known. Once profiles are collected, samples are *cleaned* using common data-processing techniques to remove outliers, and finally least squares regression is used to derive performance models.

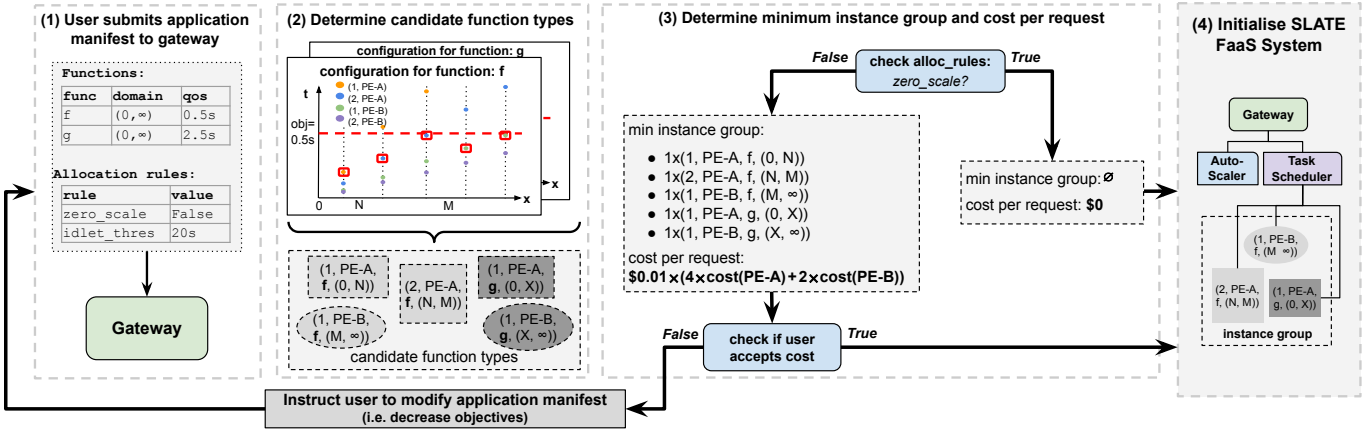

Fig. 3. The four stages of SLATE FaaS Configuration (Section II-C).

C. Configuration

The configuration phase is a crucial and a novel aspect of our approach, designed to reduce decision-making overhead at execution time. Fig. 3 depicts the four key stages in configuration of a SLATE FaaS system:

(1) User submits the *application manifest*: The application manifest lays out the user's requirements, including a list of functions in the application and an optional set of allocation rules. For each function, the user specifies the domain that needs to be supported and a latency (timing) objective (i.e. a maximum latency target for that function with any input size). The optional allocation rules tune the behaviour of the auto-scaler, and include the following:

a) Scale down idle time threshold (e.g. remove an instance if its idle time $>$ Ns)

b) A zero-scaling flag (i.e. $True$ if scaling to zero is allowed, $False$ if not)

(2) *Candidate function types* are determined: The list of candidate function types specifies all types considered by the auto-scaler during execution (Section II-B). Determining this list is a critical aspect of SLATE, since pruning the search space by restricting our system to a set of candidate types removes significant decision-making overhead at execution time.

Each function may have multiple implementations targeting different resource configurations, (N, PE). As previously mentioned, performance models for each available function implementation are derived offline and used to determine candidate instance types.

For this purpose, we generate a graph for each function and corresponding latency objective in the manifest, plotting predicted execution times for all implementations and inputs in the specified domain. See the example in Fig 4. The *best* resource configuration, (N, PE), for each input in the domain is determined. That is, the configuration which meets the latency objective with the minimum execution cost. Candidate types are identified for each function based on these 'best' configurations for each range of inputs: $(N, PE, f, (min, max))$. Note that changing the latency objective can affect the number of candidate function types.

In this way, the domain is automatically segmented into different sub-domains suited to different function types. So, during execution, when a function request is submitted at runtime, the task scheduler can immediately identify the most suitable instance.

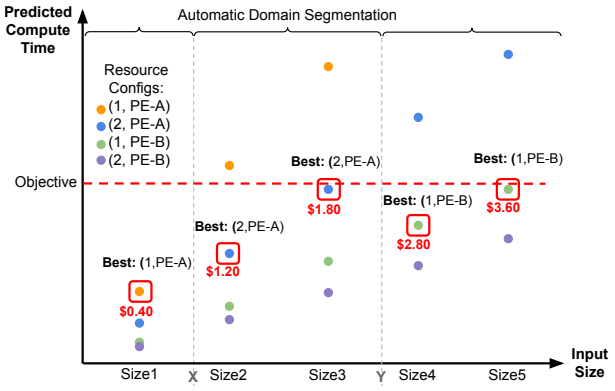

Fig. 4. During the configuration phase, users can establish the timing objective for each function. **SLATE** automatically segments the input domain and identifies the candidate instance types to be used during execution.

(3) *Minimum instance group* and request cost is determined: If zero-scaling is enabled, the minimum instance group is an empty set. If zero-scaling is disabled, the minimum resource group contains one instance of each candidate function type i.e. there will always be at least one instance of each candidate deployed in the instance group.

(4) A SLATE FaaS system is initialised: When a user accepts the minimum group cost, the system is initialised: a function instance is created for each type in the starting group if zero-scaling is disabled; the *task scheduler* is initialised with access to the instance group; and the *auto-scaler* is initialised with access to the instance group, the candidate types, and the allocation rules.

III. Evaluation

A. Experimental Setup

We validate our heterogeneous FaaS approach by simulating our **SLATE** architecture using empirically derived performance models for various case-study applications. We consider applications with implementations targeting multi-CPU and multi-FPGA resource configurations. Using our models, we can predict the latency of executing each application with a specified input workload on a given resource configuration. Note that although our evaluation focuses on FPGA and CPU resources, our approach supports other types of processing elements, such as GPUs, as long as implementations and associated performance models are available.

We currently support three case-study functions in different domains: (1) AdPredictor [8], advertisement click prediction (machine learning); (2) Exact Align [9], sequence alignment (bioinformatics); and (3) N-body Simulation [10], particle simulation (physics). These case studies are examples of HPC applications that are not well-supported by current cloud platforms (PaaS and FaaS). We consider optimised multi-CPU and multi-FPGA implementations for each, developed to target a platform with 12 CPU cores and 24 Max4 Dataflow Engines (DFEs) [11]. A DFE is a complete compute device system developed by Maxeler [12], which contains an FPGA as the computation fabric, RAM for bulk storage, logic to connect the device to a CPU host, and all necessary interfaces, interconnects, and circuitry. Our CPU implementations are programmed in C++, while the DFE implementations are written in MaxJ, a Java DSL used to describe dataflow programs.

We set the following pricing model for our simulations: 1 CPU-s costs \$0.00002 and 1 DFE-s costs \$0.00008. 1 CPU-s corresponds to an execution for one second on a CPU, conversely 1 DFE-s corresponds to an execution for one second on a DFE. Each request costs $\$(1\% * min\_group\_cost)$. So, for 10,000 requests, each executed for 10s on $(2, DFE)$ configurations, with a minimum group containing a $(2, DFE)$ instance and a $(1, CPU)$ instance, the user would be charged:

- Request cost: $10^4 \times (2 \times \$0.00008 + 1 \times \$0.00002) = \$1.80$
- Execution cost: $10^4 \times 10s \times (2 \times \$0.00008) = \$16.00$
- Total cost: \$17.80

This model is based on the FaaS pricing for AWS Lambda [2], where the request cost is \$0.0000003, and the average execution cost is \$0.00002 per second when serviced by one CPU instance with 1 GB of RAM. Our DFE cost is based on AWS EC2 FPGA-optimised instances compared to general-purpose CPU instances, where a `f1.2xlarge` instance costs roughly 4 times more than an `m4.2xlarge` instance [1].

To validate our empirical performance models and thus our simulation results, we compare observed execution latency to model-predicted latency for various tasks. The average model errors are included in Table I. In general, the observed error is less than 10%, but up to a maximum of 12.3% for Exact Align DFE implementations due to latency variations observed for the execution of the same tasks.

TABLE I
Average error in performance models

Application and Resource Configuration	Average Error
Exact Align ([1,2,4,6,8], CPU)	0.3%
Exact Align ([1,2,4,8], DFE)	12.3%
AdPredictor ([1,2,4,6,8], CPU)	0.9%
AdPredictor ([1,2,4,8], DFE)	1.4%
N-Body Sim ([1,2,4,6,8], CPU)	6.1%
N-Body Sim (1,DFE)	2.7%

B. Identifying Candidate Function Types

In order to compare **SLATE** heterogeneous function groups to homogeneous function groups, we first need to derive candidate function types which will be employed during execution. Using our performance models and the approach outlined in Section II-C, we generate the graphs in Fig. 5 to identify candidate types for each case-study application's input domain according to latency requirements (see Table II).

As previously explained, **SLATE** automatically segments the domain to classify inputs corresponding to the function type they are suited to. For instance, if we set an objective of 300ms for every Exact Align task, **SLATE** automatically identifies three sub-domains (task types) and the function types that will execute them, namely: $\boxed{s}$ (*small*) tasks are suited to $(1, CPU, align, s)$ functions, $\boxed{m}$ (*medium*) tasks are suited to $(1, DFE, align, m)$ functions, and $\boxed{l}$ (*large*) tasks are suited to $(2, DFE, align, l)$ functions.

Based on these candidate function types, we run experiments using the function groups outlined in Table II. For each case-study, we consider:

a) A heterogeneous **SLATE** function group: with heterogeneous candidate types determined in Fig. 5.
b) A homogeneous CPU function group: suited to $\boxed{s}$ traffic.
c) A homogeneous DFE function group: suited to $\boxed{l}$ traffic.

Note that for N-Body simulation, there is one function type which is best for all workloads, $(1, DFE, nbody, \{s, l\})$, hence, we do not consider a homogeneous CPU function type for our N-Body Simulation experiments.

TABLE II
Heterogeneous and Homogeneous Function Groups

	Functions	Candidate Types	Cost of 10^6 Requests
SLATE	Exact Align (Obj=300ms)	$(1, CPU, align, s)$ $(1, DFE, align, m)$ $(2, DFE, align, l)$	\$0.26
	AdPredictor (Obj=120ms)	$(1, CPU, adp, s)$ $(1, DFE, adp, l)$	\$0.10
	N-Body Sim. (Obj=400ms)	$(1, DFE, nbody, \{s, l\})$	\$0.08
Homog. CPU	Exact Align	$(1, CPU, align, \{s, m, l\})$	\$0.02
	AdPredictor	$(1, CPU, adp, \{s, l\})$	\$0.02
Homog. DFE	Exact Align	$(2, DFE, align, \{s, m, l\})$	\$0.16
	AdPredictor	$(1, DFE, adp, \{s, l\})$	\$0.08
	N-Body Sim.	$(1, DFE, nbody, \{s, l\})$	\$0.08

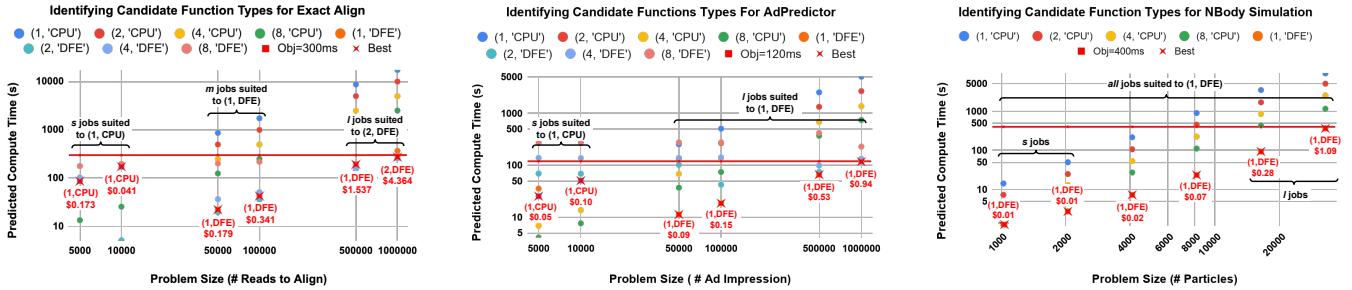

Fig. 5. Determining candidate function types for each case-study: **Exact Align** $((1, CPU, align, s), (1, DFE, align, m), (2, DFE, align, l))$, **AdPredictor** $((1, CPU, adp, s), (1, DFE, adp, l))$, and **N-body Simulation** $((1, DFE, nbody, \{s, l\}))$.

C. Performance Evaluation

To evaluate the performance of **SLATE** heterogeneous functions, we compare the execution latency of an individual task with a **SLATE**-selected function instance to each homogeneous function instance in Table II. The latency using **SLATE** functions takes into consideration the overhead of the task scheduler selecting an instance type. In practice, this overhead was observed to be on the order of $1\mu s$. This overhead is negligible due to the offline and configuration stages, which allows the task scheduler to perform one-to-one mapping decisions at runtime.

Table III presents the speedup of execution using **SLATE** compared to employing homogeneous instances, as well as the corresponding improvements in cost. Homogeneous function instances achieve the same latency and execution cost as **SLATE** for task types to which they are suited. That is, $\boxed{s}$ AdPredictor and Exact Align tasks executed on homogeneous CPU instances, $\boxed{l}$ AdPredictor and Exact Align tasks executed on homogeneous DFE instances, and all N-Body Simulation tasks executed on homogeneous DFE instances. That is, 1.0 times speedup and cost decrease.

For task types to which homogeneous instances are not suited, the latency differs from the **SLATE**-selected instances, and **SLATE** is more cost effective. That is, $\boxed{s}$ AdPredictor and Exact Align tasks executed on homogeneous DFE instances, and $\boxed{l}$ AdPredictor and Exact Align tasks executed on homogeneous CPU instances. In these cases, whether the latency is greater or less than the **SLATE**-selected instance, execution is more costly. For instance, for $align(5000)$, **SLATE** incurs a *slowdown* (1/0.7 = 1.4 times) but achieves a much greater corresponding cost decrease (5.5 times). Since the **SLATE**-selected instance is guaranteed to meet a specified latency objective, it is sufficiently performant and more cost effective than any homogeneous instance.

In general, since **SLATE** is able to tune instance type selection individually for every task, **SLATE** functions achieve the most cost effective performance overall. For functions with multiple task types, homogeneous function instances will only achieve cost effective performance for one task type.

TABLE III
LATENCY SPEEDUP AND EXECUTION COST DECREASE OF **SLATE** COMPARED TO HOMOGENEOUS FUNCTIONS FOR DIFFERENT TASKS

Task	*type*	**SLATE Speedup (Cost Improvement)**	
		Homog. CPU	**Homog. DFE**
align(5000)	$\boxed{s}$	1.0× (1.0×)	0.7× (5.5×)
align(10000)	$\boxed{s}$	1.0× (1.0×)	0.7× (5.5×)
align(500000)	$\boxed{l}$	59.0× (7.4×)	1.0× (1.0×)
align(1000000)	$\boxed{l}$	63.5× (7.9×)	1.0× (1.0×)
adp(5000)	$\boxed{s}$	1.0× (1.0×)	1.4× (5.6×)
adp(10000)	$\boxed{s}$	1.0× (1.0×)	0.7× (2.8×)
adp(500000)	$\boxed{l}$	38.4× (9.6×)	1.0× (1.0×)
adp(1000000)	$\boxed{l}$	43.2× (10.8×)	1.0× (1.0×)
nbody(4096)	$\boxed{s}$	-	1.0× (1.0×)
nbody(8192)	$\boxed{s}$	-	1.0× (1.0×)
nbody(32768)	$\boxed{l}$	-	1.0× (1.0×)
nbody(65536)	$\boxed{l}$	-	1.0× (1.0×)

D. Cost Efficiency Evaluation

To evaluate the cost efficiency of our approach, we compare the costs of executing sequences of 1000 tasks using **SLATE** functions to each homogeneous function group in Table II, where the fixed cost for 1 million requests is included in the last column.

As mentioned, FaaS pricing models include an execution cost, based on the duration of the task, as well as a fixed cost per request. Since our approach automatically selects function instances that are the most cost effective for each task, the improvements in execution cost are implicit, as explained in the previous section. However, per our pricing model (Section III-A), heterogeneous function groups with multiple candidate workers typically have higher fixed request costs than homogeneous groups. We therefore need to consider the total cost of executing sequences of multiple tasks to effectively compare the cost efficiency of **SLATE** to homogeneous functions groups.

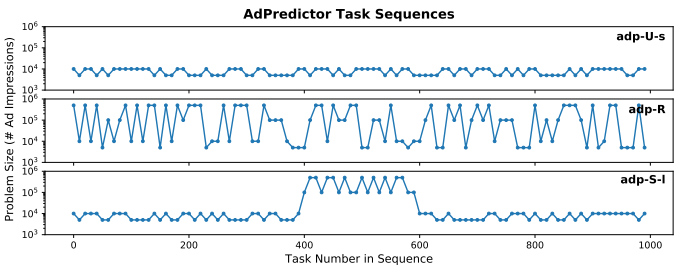

Fig. 6. Examples of uniform, random, and spiked task sequences (AdPredictor uniform $\boxed{s}$, random, and spike of $\boxed{l}$). 100 samples are plotted for each.

TABLE IV
COST IMPROVEMENT OF **SLATE** FUNCTIONS COMPARED TO HOMOGENEOUS FUNCTIONS FOR DIFFERENT TASK SEQUENCES

Function	Traffic Type	SLATE Cost Improvement	
		Homog. CPU	Homog. DFE
Exact Align	uniform $\boxed{s}$	0.5×	3.1×
	uniform $\boxed{l}$	7.2×	1.0×
	random	6.4×	1.2×
	spike of $\boxed{s}$	7.0×	1.0×
	spike of $\boxed{l}$	4.8×	1.7×
AdPredictor	uniform $\boxed{s}$	0.5×	2.1×
	uniform $\boxed{l}$	9.2×	1.0×
	random	7.7×	1.2×
	spike of $\boxed{s}$	8.7×	1.0×
	spike of $\boxed{l}$	5.2×	1.5×
N-Body Sim.	uniform $\boxed{s}$	-	1.0×
	uniform $\boxed{l}$	-	1.0×
	random	-	1.0×
	spike of $\boxed{s}$	-	1.0×
	spike of $\boxed{l}$	-	1.0×

For each function, we consider $\boxed{s}$ or $\boxed{l}$ task types and sequences with *uniform* traffic (1000 tasks of the same type), *random* traffic (a random sequence with 500 tasks of each type), and *spiked* traffic (mostly one type with a spike of 200 of the other type). Examples of these traffic types are depicted in Fig. 6 for AdPredictor. The cost decrease achieved by **SLATE** compared to each homogeneous function group is included in Table IV, where a value < 1 indicates a cost increase.

For uniform sequences with tasks to which homogeneous instances are suited, the homogeneous groups are equally or more cost effective than **SLATE**. For instance, uniform $\boxed{s}$ AdPredictor and Exact Align sequences executed on homogeneous CPU instances are 2 times less expensive than **SLATE**, while uniform $\boxed{l}$ AdPredictor and Exact Align sequences and all uniform N-Body Simulation sequences executed on homogeneous DFE instances are equal in cost to **SLATE**. In the cases where there is homogeneous $\boxed{s}$ traffic, the significant reduction in fixed costs by using homogeneous instance groups leads to a reduction in overall cost of the sequence.

For uniform sequences with tasks to which homogeneous instances are not suited, the homogeneous groups are more costly than **SLATE** due to increased latency and thus execution costs as explained in the previous section. **SLATE** costs 7.2 times less than homogeneous CPU functions for uniform $\boxed{l}$ Exact Align traffic, and 3.1 times less than homogeneous DFE functions for uniform $\boxed{s}$ Exact Align traffic.

For non-uniform task sequences with heterogeneous traffic (random or spiked), **SLATE** is more or equally cost effective than the homogeneous groups in all cases. **SLATE** costs 8.7 times less than homogeneous CPU functions for AdPredictor traffic with a spike of $\boxed{s}$ tasks, and 1.5 times less than homogeneous DFE functions for AdPredictor traffic with a spike of $\boxed{l}$ jobs. The only cases where **SLATE** is equal in cost to the homogeneous DFE group (i.e. does not show an improvement) is for AdPredictor or Align sequences with a spike of $\boxed{s}$ tasks amongst mostly $\boxed{l}$ tasks, since the prevalence of higher latency $\boxed{l}$ jobs dominates the overall cost.

E. Discussion

Based on our evaluation, we expect that in scenarios with heterogeneous traffic comprised of tasks that have very different computational requirements, **SLATE** is likely to provide cost and performance benefits over homogeneous FaaS. However, in cases where there is predictable uniform traffic, it is better to use homogeneous functions with a resource configuration tuned to all traffic. For example, with N-Body Simulation, there is no benefit of using heterogeneous **SLATE** functions over using homogeneous DFE functions.

For cases with heterogeneous traffic, an expert user may determine function types best suited to each traffic type, and deploy separate homogeneous function groups for each type of task. This might avoid increased fixed costs of heterogeneous **SLATE** groups, but requires effort and expertise to segment traffic into types and tune instances to each. On the other hand, non-expert users are unlikely to know which instance types are best suited to each task type. Therefore, the ability of **SLATE** to automatically identify suitable candidate function types and to segment function domains accordingly is beneficial to both experts (saving effort) and non-experts alike.

Finally, our simulation calculations do not currently take into account the overhead of initialisation and spawning new function instances (including dynamic reconfiguration), however we applied the same assumption to both heterogeneous and homogeneous groups in our evaluation. We intend to study the mechanisms for reducing this spawning overhead, for instance, by pre-allocating instances according to traffic patterns, in future work.

IV. RELATED WORK

Table V summarises the current, key approaches in managed PaaS and FaaS cloud systems. Commercial PaaS frameworks (Microsoft Azure App Services [6], Google AppEngine [7], and AWS Elastic Beanstalk [13]) automatically manage and elastically scale application resources. However, they have limited support for hardware accelerators and heterogeneity. Applications can only scale horizontally using a single type of resource. ORIAN [5] extends the PaaS execution model to support hardware accelerators and heterogeneity. However, PaaS is resource-oriented, requiring effort in terms of application deployment as well as paying for resources even when they are idle.

These shortcomings of PaaS are addressed by FaaS, where users pay only for serviced function requests. This simplifies deployment, orchestration, and pricing. None of the three most popular commercial offerings (AWS Lambda, Microsoft Azure Functions [3], and Google Cloud Functions [4]) support hardware accelerators. They are limited to homogeneous, CPU-based function types, with users only able to select an amount of memory and CPU. Furthermore, auto-scaling is limited to replication of homogeneous function instances. **SLATE** FaaS, on the other hand, supports functions that can scale with arbitrary resource configurations and accelerator types based on user-supplied objectives and rules.

TABLE V
COMPARISON BETWEEN DIFFERENT PAAS AND FAAS APPROACHES

Approach	Type	Scaling	Service
Azure App Services [6]	Commercial	Uniform	PaaS
Google App Engine [7]	Commercial	Uniform	PaaS
AWS Beanstalk [13]	Commercial	Uniform	PaaS
ORIAN [5]	Open Source	Heterogeneous	PaaS
AWS Lambda [2]	Commercial	Uniform	FaaS
Google Functions [4]	Commercial	Uniform	FaaS
Azure Functions [3]	Commercial	Uniform	FaaS
OpenFaas [14]	Open Source	Uniform	FaaS
OpenWhisk [15]	Open Source	Uniform	FaaS
Kubeless [16]	Open Source	Uniform	FaaS
SLATE	Open Source	Heterogeneous	FaaS

Outside of commercial FaaS offerings, frameworks like OpenFaaS [14], OpenWhisk [15] and Kubeless [16] aim to provide a flexible environment in which users can build their own FaaS systems. Such systems provide greater flexibility to developers, allowing them to implement and deploy their own function types and control certain resource management mechanisms. However, they are still limited to homogeneous auto-scaling, and employing a single instance type.

Finally, there are various research projects dedicated to making accelerators and other heterogeneous resources available in the cloud. For instance, the work of [17] virtualises FPGAs by mapping accelerators to regions of FPGAs managed by runtime managers on local processors. FPGAVirt [18] proposes a hardware/software co-design framework focused on abstraction, sharing, and isolation using an overlay to divide FPGAs into 'virtual functions' with a management service that maps computations to available regions. Our **SLATE** work can make use of these virtualisation efforts on hardware accelerators, as long as performance is isolated and deterministic, in order to make effective use of performance models.

V. CONCLUSION

We present **SLATE**, a fully-managed Function-as-a-Service (FaaS) system for deploying serverless functions onto heterogeneous cloud infrastructures. **SLATE** improves the accessibility of specialised, heterogeneous resources to cloud tenants by extending the traditional homogeneous FaaS execution model to support heterogeneous functions, with abstracted, automatic runtime management of multiple resource types, such as CPUs and FPGAs. We have simulated our approach, considering case study functions in three application domains (machine learning, bio-informatics, and physics), with implementations targeting FPGA and CPU resource configurations. Compared to homogeneous CPU and FPGA functions, simulation results achieve respectively a cost improvement for non-uniform task traffic of up to 8.7 times and 1.7 times, while maintaining individual task latency (timing) objectives.

Current and future work includes extending our simulator into a full prototype, supporting a compilation path to allow user-defined computations to be managed, and targeting other application domains and accelerator types, such as GPUs and application-specific devices.

ACKNOWLEDGEMENT

The support of Intel and the U.K. EPSRC (grants EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1 and EP/L00058X/1) is gratefully acknowledged.

REFERENCES

[1] Amazon Web Services, "Amazon EC2," [Online; accessed Apr-2020]. [Online]. Available: {https://aws.amazon.com/ec2/}

[2] ——, "AWS Lambda: Serverless Compute," [Online; accessed Apr-2020]. [Online]. Available: {https://aws.amazon.com/lambda/}

[3] Microsoft Azure, "Azure Functions: Serverless Compute," [Online; accessed Apr-2020]. [Online]. Available: {https://azure.microsoft.com/en-gb/services/functions/}

[4] Google Cloud Platform, "Cloud Functions," [Online; accessed Apr-2020]. [Online]. Available: {https://cloud.google.com/functions}

[5] J. Vandebon, J. G. F. Coutinho, W. Luk, E. Nurvitadhi, and M. Naik, "Enhanced Heterogeneous Cloud: Transparent Acceleration and Elasticity," in *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, pp. 162–170.

[6] "Azure App Service." [Online]. Available: {https://azure.microsoft.com/en-gb/services/app-service/}

[7] "Google App Engine." [Online]. Available: {https://cloud.google.com/appengine/}

[8] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich, "Web-scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine," in *ICML*. USA: Omnipress, 2010, pp. 13–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104326

[9] J. Arram, T. Kaplan, W. Luk, and P. Jiang, "Leveraging FPGAs for Accelerating Short Read Alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 14, no. 3, pp. 668–677, May 2017.

[10] Maxeler, "N-Body Particle Simulation," https://github.com/maxeler/NBody, 2015, [Online; accessed Jan-2020].

[11] "Maxeler MPC-X Series," [Online; accessed Apr-2020]. [Online]. Available: {https://www.maxeler.com/products/mpc-xseries/}

[12] "Maxeler Technologies," [Online; accessed Apr-2020]. [Online]. Available: {https://www.maxeler.com/}

[13] "AWS Elastic Beanstalk." [Online]. Available: {https://aws.amazon.com/elasticbeanstalk/}

[14] "OpenFaaS Introduction: Serverless Functions Made Simple," [Online; accessed Apr-2020]. [Online]. Available: {https://docs.openfaas.com/}

[15] Apache Software Foundation, "Open Source Serverless Cloud Platform," [Online; accessed Apr-2020]. [Online]. Available: {https://openwhisk.apache.org/}

[16] Kubeless, "The Kubernetes Native Serverless Framework," [Online; accessed Apr-2020]. [Online]. Available: {https://kubeless.io/}

[17] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[18] J. Mbongue, F. Hategekimana, D. T. Kwadjo, D. Andrews, and C. Bobda, "FPGAVirt: A Novel Virtualization Framework for FPGAs in the Cloud," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018.