HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D2.5 Reference Implementation of Architectural Building Blocks

Due date of deliverable: 30-06-2022
Actual submission date: 20-07-2022

Start date of project: 01-01-2019 Duration: 42 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	98
WP/Task related to this document	WP2 / T2.1, T2.3, T2.4, T2.5, T2.6
WP/Task responsible	URV
Leader	Pedro García Lopez (URV)
Technical Manager	Pedro García Lopez (URV)
Quality Manager	Marc Sanchez (URV)
Author(s)	Pedro García (URV), Rut Palmero (ATOS), Lara Lopez (ATOS), Theodore Alexandrov (EMBL), Paolo Ribeca (JHI), Carlos Segarra (IMP), Guo Li (IMP), Vasily Sartakov (IMP), Marc Sánchez (URV)
Partner(s) Contributing	URV, ATOS, EMBL, JHI.
Document ID	CloudButton_D2.5_Public.pdf
Abstract	The deliverable aims to present the final design and reference implementation of the CloudButton toolkit. It also includes the performance evaluation study on the data and experiments from the different use cases.
Keywords	use cases, experiments, architecture, state of the art.

History of changes

Version	Date	Author	Summary of changes
0.1	09-05-2022	Pedro García (URV)	Project Vision, Towards Transparency, CloudButton Architecture
0.2	27-05-2022	Carlos Segarra (IMP), Simon Shillaker (IMP)	Genomics use case integration with Faasm.
0.3	30-05-2022	Carlos Segarra (IMP), Guo Li (IMP), Vasily Sartakov (IMP)	Update transparency section for Faasm
0.4	17-07-2022	Paolo Ribeca (JHI)	Genomics use case
0.5	17-07-2022	Aitor Arjona (URV)	Geospatial use case
0.6	17-07-2022	Theodore Alexandrov (EMBL)	Metabolomics use case
Final	18-07-2022	Pedro Garcia Lopez (URV)	Final edit including introduction, conclusions, and final updates in different sections.

Table of Contents

1 Executive summary

The deliverable D2.3 "Serverless Compute Engine Reference Implementation" aims to present the final design and reference implementation of the CloudButton toolkit. It also includes the performance evaluation study on the data and experiments from the different use cases.

This deliverable is complemented by the public github repositories produced in the project. In particular, we outline here the following ones:

- Lithops (`https://github.com/lithops-cloud/lithops`)
- Crucial (`https://github.com/crucial-project/crucial`)
- Faasm (`https://github.com/faasm/faasm`)
- Serverless Metabolomics Use Case (`https://github.com/metaspace2020/Lithops-METASPACE`)
- Serverless Geospatial Use Case (`https://github.com/cloudbutton/geospatial-usecase`)
- Serverless Genomics Use Case (`https://gitlab1.bioss.ac.uk/lmarcello/serverless_genomics/-/tree/main/variant_caller`)

Lithops, Crucial and FaasM are the three core contributions of the CloudButton toolkit. Focusing on exploitation and production ready environments, Lithops is the major technical achievement of the project. It is a Serverless Data Analytics platform used today in production in EMBL (METASPACE), IBM, Hutton and many other locations.

In the cloudbutton.eu web site we also offer the final reference implementation of all software components. It also includes tutorials to facilitate the adoption of the platform by third-party developers.

This document is structured as follows. Section 2 provides a brief introduction to the deliverable. Section 3 presents the project vision and an analysis of Serverless Analytics State of the Art. In Section 4, we present our reflections on several paths to achieve transparency in disaggregated systems. Section 5 describes the project Architecture and main building blocks. Sections 6-8 are devoted to the description of the use cases. Finally, Section 9 contains the conclusions of this document.

2 Introduction

The CloudButton project started in January 2019 and it finished in June 2022. CloudButton aimed to create a novel Serverless Dara Analytics Platform that could benefit from the promising services of the emerging serverless computing paradigm.

In particular, serverless, fully-managed experience could contribute to simplify the deployment of Big Data analytics pipelines in the Cloud. Our major aim was to considerably simplify Big Data computing in the Cloud thanks to serverless technologies. As starting point, we were inspired by the seminal paper by Jonas et al. [1], where it was stated the following enlightening phrase: *"Why is there no cloud button?"* The authors outlined how their students from Berkeley simply wished they could easily push a button and have their code — existing, optimized, single-machine code — running on the cloud".

After three years, we are proud to say that we have realized the CloudButton vision, presenting and validating the vision of transparency [2] [3]. We defined transparency as *"running unmodified single-machine code over effectively unlimited compute, storage, and memory resources in the Cloud"*. For this reason, it is not rare that our first Key Performance Indicator (KPI) to demonstrate the success of the project have been **Simplicity & Productivity**.

Simplicity & Productivity. The project has produced three main efforts in transparently moving applications to the Cloud: Python, Java, and C++. The tree of them contributed to the Simplicity & Productivity KPI by overly simplifying the deployment and execution of those applications in the Cloud. These are:

- **Python:** Lithops can be sensed as the key Python toolkit that demonstrated transparency for Python MapReduce and `multiprocessing` applications, such as Deep Learning (DL), Evolution Strategies, and Proximal Policy Optimization, to name a few;
- **Java:** Crucial and `sshell` is the major toolkit to demonstrate transparency for Java applications (e.g., serverless `ExecutorService`, Unbalanced Tree Search algorithm, SMILE machine learning (ML) library); and
- **C/C++:** Faasm is the major toolkit based on WebAssembly (Wasm) technology to transparently port, mostly C++ and HPC applications built upon OpenMP and OpenMPI.

We will demonstrate in this deliverable how easy is to analyze massive datasets in the Cloud with a few lines of Python code. We also provide the results of an open questionnaire that was distributed in the Lithops user community.

Performance in Big Data pipelines. A second important goal of this project is to test if serverless Cloud technologies can be suitable for executing large scale Big Data pipelines. We selected three domains involving large unstructured data: *genomics* (text files such as FAST or FASTQ data types), *metabolomics* (images such as imlMZ data types), and *geospatial* data (images from Sentinel satellite, cloud of points in LIDAR data type).

Accordingly, the second KPI to demonstrate this project revolves around **Scalability & Elasticity & Performance**. Elasticity and auto-scaling are novel features claimed by the serverless paradigm. Nevertheless, an important question was if storage and compute resources in the Cloud could sustain the stringent scaling requirements of Big Data volumes.

Just to illustrate, we have benchmarked state of the art public serverless services[1] [4] to verify that Cloud object storage can provide huge aggregated bandwidth to parallel processes (around 100GB/s) and offer thousand of cores (vCPUs) in an elastic way (1000 parallel vCPUs/s spawned in less than 100ms). We will also demonstrate in this deliverable Performance/scalabily KPIs using Big Data pipelines in the three use cases.

Another key contribution of the CloudButton project is its data-driven approach to *automatically* provision data resources based on data volumes and data types. To wit, Lithops has been enhanced to

[1] `https://github.com/lithops-cloud/applications/tree/master/benchmarks`

automatically launch the required computing devices from a plethora of data partitioning schemes developed in this project for the different data types. We considerably simplified ETL (EXtract/-Transform/Load) and preprocessing phases in the Cloud by implementing *dynamic* data partitioning schemes specially designed for Cloud object storage.

In this final deliverable, we contribute the overall vision of transparency, describe in-depth the Lithops toolkit, and validate all KPIS using the three use cases.

3 Project Vision

Current practice shows that the FaaS model is well suited for many types of applications, provided that they require a small amount of storage and memory. Indeed, this model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier (e.g., uploading a photo to Amazon S3), which encompasses many common tasks in Cloud applications. What was unclear is whether or not this new computing model could also be useful to execute data analytics applications.

This question was answered partially in 2017 with the appearance of two relevant research articles [5] [1] that demonstrated that Function-as-a-Service (FaaS) could sustain massively parallel computations in the Cloud. The former presented ExCamera, providing on-the-fly video encoding over thousands of Amazon Lambda Functions. ExCamera proved to be 60% faster and 6x cheaper than using VM instances. Another relevant work is the "Occupy the Cloud" paper, showcasing simple MapReduce jobs executed over Lambda Functions in their PyWren prototype. In this case, PyWren was 17% slower than PySpark running on `r3.xlarge` VM instances. But the authors claimed that the simplicity of configuration and inherent elasticity of Lambdas justified the performance penalty. In this paper, they did not compare the costs between their lambdas and the VM experiments.

Both research works demonstrated the enormous potential of serverless data analytics. The two major advantages are clearly the simplicity and the massive scalability and elasticity of the model. On the one hand, the scaling, deployment, provisioning, fault-tolerance, and monitoring of functions is delegated to the cloud provider. Furthermore, the programming simplicity of functions clearly paves the way to a smooth Cloud transition. On the other hand, the transparent and almost infinite elasticity boosts the analysis of huge data volumes accessible in Cloud Object Stores.

But Serverless Computing is nowadays not adequate for many data analytics tasks due to two fundamental problems: high cost and lack of performance compared to Cluster Computing or even VMs running Spark. Two recent articles have outlined the major limitations of the Serverless model in general: [6] and [7]. In the latter, they review the performance and cost of several data analytics applications. They show that MapReduce Sort (100TB) was 1% faster than VMs, but costing 15% higher; Linear Algebra (NumPyWren) was 3x slower than an MPI implementation in a dedicated cluster, but only valid for large problem sizes; and Machine Learning pipelines (Cirrus) were 3x-5x faster than VM instances, but up to 7x higher total cost.

Furthermore, existing approaches must rely on auxiliary Serverful services to circumvent the limitations of the stateless serverless model. PyWren uses Amazon S3 for storage, coordination, and communication, Locus uses Redis ElastiCache In-memory system, ExCamera relies on a external rendezvous and communication service, or Cirrus relies on disaggregated in-memory servers.

In deliverable D2.1 [8], we identified the hybrid applications combining Serverless and Serverful services as **ServerMix**, and we showed how most related work can be classified under the ServerMix umbrella term.

We also identified three important tradeoffs of the Serverless model and how these tradeoffs could be relaxed to obtain more performance:

1. Relaxing disaggregation: Using locality in memory or function placement could boost performance. Moving from a serverless data-shipping model to benefit from computation close to the data could easily achieve performance improvements. But disaggregation is the fundamental pillar of elasticity in the Cloud.

2. Relaxing isolation: Co-locating related functions (namespaces) in the same containers, and reusing containers could also improve performance. Providing direct communication between functions could also facilitate shared replicated memory models. Leveraging lightweight containers or even using language-level constructs would also reduce cold starts and boost inter-function communications. But isolation is the basis for multi-tenancy and security.

3. Flexible QoS and scheduling: To ensure SLAs it could be possible to implement more sophisticated scheduling algorithms that can reserve resources or entire nodes to functions, or even

execute them in specialized hardware like GPUs. But simple location-agnostic scheduling is the basis for reduced start times and increased cloud resource utilization.

We claimed that the so-called "limitations" of the serverless model are indeed its defining traits. When applications should require aggregation (computation close to the data), relaxing isolation (co-location, direct communication), or tunable scheduling (predictable performance, hardware acceleration) a suitable solution is to build a ServerMix.

Therefore, in CloudButton project we advocate for (i) Smart Scheduling as a mechanism for providing transparent provisioning to applications while optimizing the cost-performance tuple in the Cloud, (ii) Fine-grained State Disaggregation thanks to Serverless Mutable Consistent State services, and (ii) Lightweight and Polyglot Serverful Isolation: novel lightweight Serverful FaaS runtimes based on WebAssembly as universal multi-language substrate.

3.1 Serverless Analytics State of the Art

Despite the stringent constraints of the FaaS model, a number of works have managed to show how this model can be exploited to process and transform large amounts of data [1, 9, 10], encode videos [5], and run large-scale linear algebra computations [11], among other applications. Surprisingly, and contrary to intuition, most of these serverless data analytics systems are indeed good *ServerMix* examples, as they combine both serverless and serverful components.

In general, most of these systems rely on a external, serverful provisioner component [1, 9, 10, 5, 11]. This component is in charge of calling and orchestrating serverless functions using the APIs of the chosen cloud provider. Sometimes the provisioner is called “coordinator” (e.g., as in ExCamera [5]) or “scheduler” (e.g., as in Flint [10]), but its role is the same: orchestrating functions and providing some degree of fault tolerance. But the story does not end here. Many of these systems require additional serverful components to overcome the limitations of the FaaS model. For example, recent works such as [12] use disaggregated in-memory systems such as ElastiCache Redis to overcome the throughput and speed bottlenecks of slow disk-based storage services such as S3. Or even external communication or coordination services to enable the communication among functions through a disaggregated intermediary (e.g., ExCamera [5]).

To fully understand the different variants of *ServerMix* for data analytics, we will review each of the systems one by one in what follows. Table 1 details which components are serverful and serverless for each system.

PyWren [1] is a proof of concept that MapReduce tasks can be run as serverless functions. More precisely, PyWren consists of a serverful function scheduler (i.e., a client Python application) that permits to execute “map” computations as AWS Lambda functions through a simple API. The “map” code to be run in parallel is first serialized and then stored in Amazon S3. Next, PyWren invokes a common Lambda function that deserializes the “map” code and executes it on the relevant datum, both extracted from S3. Finally, the results are placed back into S3. The scheduler actively polls S3 to detect that all partial results have been uploaded to S3 before signaling the completion of the job.

Lithops [9] is a PyWren derived project which adapts and extends PyWren for multiple cloud providers, such as IBM CLoud, Azure or Google Cloud. It includes a number of new features, such as broader MapReduce support, automatic data discovery and partitioning, integration with Jupiter notebooks, and simple function composition, among others. For function coordination, Lithops uses RabbitMQ to avoid the unnecessary polling to the object storage service (IBM COS), thereby improving job execution times compared with PyWren.

ExCamera [5] performs digital video encoding by leveraging the parallelism of thousands of Lambda functions. Again, ExCamera uses serverless components (AWS Lambda, Amazon S3) and serverful ones (coordinator and rendezvous servers). In this case, apart from a coordinator/scheduler component that starts and coordinates functions, ExCamera also needs of a rendezvous service, placed in an EC2 VM instance, to communicate functions amongst each other.

Stanford’s gg [13] is an orchestration framework for building and executing burst-parallel applications over Cloud Functions. gg presents an intermediate representation that abstracts the compute

Table 1: *ServerMix* applications

Systems	**Components**	
	Serverful	***Serverless***
PyWren [1]	Scheduler	AWS Lambda, Amazon S3
PyWren-IBM-Cloud [9]	Scheduler	IBM Cloud Functions, IBM COS, RabbitMQ
ExCamera [5]	Coordinator and rendezvous servers (Amazon EC2 VMs)	AWS Lambda, Amazon S3
gg [13]	Coordinator	AW Lambda, Amazon S3, Redis
Flint [10]	Scheduler (Spark context on client machine)	AW Lambda, Amazon S3, Amazon SQS
Numpywren [11]	Provisioner, scheduler (client process)	AWS Lambda, Amazon S3, Amazon SQS
Cirrus [7]	Scheduler, parameter servers (large EC2 VM instances with GPUs)	AWS Lambda, Amazon S3
Locus [12]	Scheduler, Redis service (AWS ElastiCache)	AWS Lambda, Amazon S3

and storage platform, and it provides dependency management and straggler mitigation. Again, gg relies on an external coordinator component, and an external Queue for submitting jobs (gg's thunks) to the execution engine (functions, containers).

Flint [10] implements a serverless version of the PySpark MapReduce framework. In particular, Flint replaces Spark executors by Lambda functions. It is similar to PyWren in two main aspects. On one hand, it uses an external serverful scheduler for function orchestration. On the other hand, it leverages S3 for input and output data storage. In addition, Flint uses the Amazon's SQS service to store intermediate data and perform the necessary data shuffling to implement many of the PySpark's transformations.

Numpywren [11] is a serverless system for executing large-scale dense linear algebra programs. Once again, we observe the *ServerMix* pattern in numpywren. As it is based in PyWren, it relies on a external scheduler and Amazon S3 for input and ouput data storage. However, it adds an extra serverful component in the system called provisioner. The role of the provisioner is to monitor the length of the task queue and increase the number of Lambda functions (executors) to match the dynamic parallelism during a job execution. The task queue is implemented using Amazon SQS.

Cirrus machine learning (ML) project [7] is another example of a hybrid system that combines serverful components (parameter servers, scheduler) with serverless ones (AWS Lambda, Amazon S3). As with linear algebra algorithms, ML researchers have traditionally used clusters of VM instances for the different tasks in ML workflows. Nonetheless, a fixed a cluster size can either lead to severe underutilization or slowdown, since each stage of a workflow can demand significantly different amounts of resources. Cirrus addresses this challenge by enabling every stage to scale to meet its resource demands by using Lambda functions. The main problem with Cirrus is that many

ML algorithms require state to be shared between cloud functions, for it uses VM instances to share and store intermediate state. This necessarily converts Cirrus into another example of a *ServerMix* system.

Finally, the most recent example of *ServerMix* is Locus [12]. Locus targets one of the main limitations of the serverless stateless model: data shuffling and sorting. Due to the impossibility of function-to-function communication, shuffling is ill-suited for serverless computing, leaving no other choice but to implement it through an intermediary cloud service, which could be cost prohibitive to deliver good performance. Indeed, the first attempt to provide an efficient shuffling operation was realized in PyWren [1] using 30 Redis ElastiCache servers, which proved to be a very expensive solution. The major contribution of Locus was the development of a hybrid solution that considers both cost and performance. To achieve an optimal cost-performance trade-off, Locus combined a small number of expensive fast Redis instances with the much cheaper S3 service, achieving comparable performance to running Apache Spark on a provisioned cluster.

We did not include SAND [14] in the list of *ServerMix* systems. Rather, it proposes a new FaaS runtime. In the article, the authors of SAND present it as an alternative high-performance serverless platform. To deliver high performance, SAND introduces two relaxations in the standard serverless model: one in *disaggregation*, via a hierarchical message bus that enables function-to-function communication, and another in *isolation*, through application-level sandboxing that enables packing multiple application-related functions together into the same container. Although SAND was shown to deliver superior performance than Apache OpenWhisk, the paper failed to evaluate how these relaxations can affect the scalability, elasticity and security of the standard FaaS model.

Recent works also outline the need for novel serverless services providing flexible disaggregated storage to serverless functions. This is the case of Pocket's ephemeral storage service [15], which provides auto-scaling and pay-per-use as a service to cloud functions. Similarly, [7] proposes as a future challenge the creation of high-performance, affordable, transparently provisioned storage. This work discusses two types of unaddressed storage needs: *Serverless Ephemeral Storage* and *Serverless Durable Storage*, both of which should deliver micro-second latencies, fault-tolerance, auto-scalability and transparent provisioning for multi-tenant workloads. The paper suggests that with a shared in-memory service, spare memory resources from one serverless application can be allocated to another. Finally, it also elaborates on why existing cloud services such as Redis or MemCached cannot fulfill the aforementioned storage needs. Actually, both in-memory services can be deemed as *serverful* due to their need for explicit provisioning and dedicated resources per tenant.

3.2 Serverless orchestration systems

An interesting alternative could be to use serverless orchestration services to coordinate and provision data analytics applications. The key question is whether these systems are actually well-prepared to orchestrate massively parallel computations. In this sense, our recent paper [16] compares three commercial serverless orchestration services along several axis: Amazon Step Functions, Azure Durable Functions, and IBM Composer, updating a previous analysis [17]. The conclusion of our study was that none of the platforms solve the orchestration of parallel computations with suitable performance.

We find that IBM Composer is the best solution available as of today. Its idea of using serverless functions for the orchestration logic is indeed a good decision to achieve fitting elasticity. However, its implementation of parallel executions fails to fulfill a reactive scheme by adding a synchronous external connection. Furthermore, the system falls short on allowing long-running workflows (some steps would be billed for idle time and waste resources); a direct consequence of the orchestration system being so tightly integrated in the FaaS infrastructure itself. We also believe that orchestration of serverless functions requires a reactive event-based design, avoiding double billing and blocking external invocations.

Unfortunately, as of today, the reality is that cloud applications are still inadvertently bound to the *ServerMix* model. In fact, several cloud providers are now transitioning to hybrid models combining serverless and serverful concepts. For instance, we have recently seen how some cloud providers

like Microsoft offer *ServerMix* FaaS services such as the Azure Premium Plan for running functions on dedicated machines while abstracting users from the provisioning phase. The Azure platform is even allowing users to pre-warm functions to reduce their cold starts.

3.3 Serverless container services

Hybrid cloud technologies are also accelerating the combination of serverless and serverful components. For instance, the recent deployment of Kubernetes (k8s) clusters in the big cloud vendors can help overcome the existing application portability issues in the cloud. There exists a plenty of hosted k8s services such as *Amazon Elastic Container Service* (EKS), *Google Kubernetes Engine* (GKE), and *Azure Kubernetes Service* (AKS), which confirm that this trend is gaining momentum. However, none of these services can be considered 100% "serverless". Rather, they should be viewed as a middle ground between cluster computing and serverless computing. That is, while these hosted services offload operational management of k8s, they still require custom configuration by developers. The major similarity to serverless computing is that k8s can provide short-lived computing environments like in the customary FaaS model.

But a very interesting recent trend is the emergence of the so-called serverless container services such as AWS Fargate, Azure Container Instances (ACI), and Google Cloud Run (GCR). These services reduce the complexity of managing and deploying k8s clusters in the cloud. While they offer serverless features such as flexible automated scaling and pay-per-use billing model, these services still require some manual configuration of the right parameters for the containers (e.g., compute, storage, and networking) as well as the scaling limits for a successful deployment.

These alternatives are interesting for long-running jobs such as batch data analytics, while they offer more control over the applications thanks to the use of containers instead of functions. In any case, they can be very suitable for stateless, scalable applications, where the services can scale-out by easily adding or removing container instances. In this case, the user establishes a simple CPU or memory threshold and the service is responsible for monitoring, load balancing, and instance creation and removal. It must be noted that if the service or application is more complex (e.g., a stateful storage component), the utility of these approaches is rather small, or they require important user intervention.

For example, AWS Fargate offers two models: Fargate launch type and EC2 launch type. The former is more serverless and requires less configuration. The latter gives users more control but also more responsibility. An analogous issue occurs with Google: Cloud Run vs. Cloud Run on GKE. The former is automated and uses standard vCPUs, while the latter enables customers to select hardware requirements and manage their cluster.

An important open source project related to serverless containers is CNCF's Knative. In short, Knative is backed by big vendors such as Google, IBM and RedHat, among others, and it simplifies the creation of serverless containers over k8s clusters. Knative simplifies the complexity of k8s and Istio service mesh components, and it creates a promising substrate for both PaaS and FaaS applications. GCR is based on Knative. IBM Cloud is also offering seamless Knative integration in their k8s services. Yet, it is hard to see how, in future terms, hosted Knative and k8s cloud services will reshape the current FaaS landscape, since, in its present form, have important implications on the key traits of the FaaS model such as disaggregation and scheduling.

As a final conclusion, we foresee that the simplicity of the serverless model will gain traction among users, so many new offerings may emerge in the next few years, thereby blurring the borders between both serverless and serverful models. Further, container services may become an interesting architecture for *ServerMix* deployments.

4 Towards transparency

An aspect to consider when moving existing codebases to the serverless paradigm is that of the transparency. Achieving full transparency would imply that we can compile, debug and run unmodified single-machine code over effectively unlimited compute, storage, and memory resources.

Transparency is an archetypal challenge in distributed systems that has not yet been adequately solved. Transparency implies the concealment from the user and the application programmer of the complexities of distributed systems. Coulouris et al. [18] define eight forms of transparency: access, location, concurrency, replication, failure, mobility, performance, and scalability.

But, despite all previous efforts, the problem is still open as seen in recent literature [1] and is, in fact, one of the main objectives of this project: simplifying the overall life cycle and programming model (in the application domain of big data analytics).

Waldo et al. [19] explain that the goal of merging the programming and computational models of local and remote computing is not new. They state that around every ten years "a furious bout of language and protocol design takes place and a new distributed computing paradigm is announced". They mention messages in the 70s, RPCs in the 80s, and objects in the 90s.

In every iteration, a new wave of software modernization is generated, and applications are ported to the newest and hot paradigm. Waldo et al. claim that all these iterations may be evolutionary stages to unify both local and distributed computing. But they are pessimistic, and they believe that this will not be possible because of latency, memory access, concurrency and partial failure.

That visionary paper even considers that, in the future, hardware improvements could make the difference in latency irrelevant, and that differences between local and remote memory could be masked. But they still claim that concurrency and partial failures preclude the unification of local and remote computing. Unlike an OS, they are telling us that a distributed system has no single point of resource allocation, synchronization, or failure.

But, what if novel cloud technologies could make the unification of local and remote paradigms possible? Are we close to the end of the cycles of software modernization? Can we just compile to the Serverless SuperComputer [20] imagined by Tim Wagner, inventor of AWS Lambda?

We argue in this deliverable that recent reductions in network latency [21, 22] are boosting resource disaggregation in the Cloud, which is the definitive catalyst to achieve transparency. Even if existing Cloud services are still in the millisecond range (100ms Lambda overhead, 10ms in Kafka, 5-20ms in S3), disaggregation has already fueled the creation of serverless computing services like Function as a Service, Cloud Object Storage, and messaging. If we can go down to µs RPCs [23, 24], novel opportunities for transparency will emerge [25, 22].

We believe that the full transparency for serverless computing will arrive when all resources (compute, storage, memory) can be offered in a disaggregated way with unlimited flexible scaling. This will also require a new generation of locality-aware scalable stateful services, smartly combining disaggregation and local resources. Also, we indentify and study five open research challenges required to achieve full transparency for most applications: (i) granular middleware and locality, (ii) memory disaggregation, (iii) virtualization, (iv) elastic programming models, and (v) optimized deployment.

4.1 DDC path to transparency

The DDC path is probably the most direct but also the most shocking for the distributed systems community. In line with recent industrial trends on Disaggregated Data centers (DDC) [26], it implies a distributed OS transparently leveraging disaggregated hardware resources like processing, memory or storage.

A canonical example is LegoOS: A disseminated, distributed OS for hardware resource disaggregation [27]. LegoOS exposes a distributed set of virtual nodes (vNode) to users. Each vNode is like a virtual machine managing its own disaggregated processing, memory and storage resources. LegoOS achieves transparency and backwards compatibility by supporting the Linux system call interface and Linux ABIs (Application Binary Interface), so that existing unmodified Linux applications can run on top of it. Even distributed applications that run on Linux can seamlessly run on a

LegoOS cluster by running on a set of vNodes. For example, LegoOS shows how two unmodified applications can be run in a distributed way: Phoenix (a single-node multi-threaded implementation of MapReduce) and TensorFlow.

Another relevant work is Arrakis: The Operating System is the Control Plane [28]. Arrakis comes from previous efforts aimed at optimizing the kernel code paths to improve data transfer and latency in the OS. In Arrakis, applications have direct access to virtualized I/O devices, which allows most I/O operations to bypass the kernel entirely without compromising process isolation. Arrakis virtualized control plane approach allows storage solutions to be integrated with applications, even allowing the development of higher level abstractions like persistent data structures. Even more, Arrakis control plane is a first step towards integration with a distributed data center network resource allocator.

If the OS can be extended with unbounded resources in a transparent way, distribution may no longer be needed for many applications – single-node parallel programming is sufficient. This is completely in line with the following assessment from the COST paper [29]: "You can have a second computer once you've shown you know how to use the first one". This paper presents a critique of the current research in distributed systems, and even suggests that "there are numerous examples of scalable algorithms and computational models; one only needs to look back to the parallel computing research of decades past".

COST stands in that paper for the "Configuration that Outperforms a Single Thread". They mainly compare optimized single-threaded versions of graph algorithms, with their equivalents in distributed frameworks like Spark, Naiad, GraphX, Giraph or GraphLab. For example, Naiad has a COST of 16 cores for executing PageRank on the twitter graph, which means that Naiad needs 16 cores to outperform a single-threaded version of the same algorithm in one machine.

An important reflection from this paper is that the overheads of distributed frameworks (coordination, serialization) can be extremely high just in order to justify scalability. But the COST paper is not proposing a solution to the scalability problem, since it is obvious that a single machine cannot scale enough for many algorithms.

But, what happens if we combine the COST idea with the DDC research? This is precisely what Gao et al.[26] validated in a simple experiment comparing a COST version with a COST-DDC one that relies on disaggregated memory (Infiniswap [30]). They demonstrate in this paper that the same code can overcome the memory limits thanks to disaggregation and still obtain good performance results.

DDC is openly challenging the so-called server-centric approach of development for the data center. DDC advocates claim that the monolithic server model where the server is the unit of deployment, operation, and failure is becoming obsolete. They advocate for a paradigm shift where many existing server-centric (cluster computing) approaches must yield to a more efficient way of managing resources in the Data Center.

The DDC paradigm is presenting server-centric cluster technologies as obsolete. But current mature Cloud technologies are built on top of server-centric models with commodity hardware and Ethernet networks. Hardware resource disaggregation is interesting, but it still relies on server-centric clusters for scaling. For example, LegoOS emulates disaggregated hardware components using a cluster of commodity servers. Existing OS approaches like LegoOS have still not dealt with serious distributed systems problems like scalability, consistency and security of the disaggregated resources in a multi-tenant Cloud setting.

4.2 Server-centric path to Transparency

Recent proposals to achieve transparency intercept calls at the programming language level, replacing local accesses with remote ones. For instance, in the context of the CloudButton project, Crucial [31] implements a serverless scheduler (aka., executor service) for Java. This allows Java threads to transparently run as serverless functions in the FaaS platform. Crucial also provides synchronization primitives as well as consistent mutable data structures atop a disaggregated in-memory storage layer. To date, the Crucial framework is limited to Java. Moreover, it does no support main memory

scaling, nor transparent access to storage. Both have to be defined beforehand by the programmer (e.g., by tuning the FaaS platform and relying on an object storage).

A second output of CloudButton is the serverless shell (`sshell`). The serverless shell is built atop Crucial and permits to execute *NIX commands and scripts in a FaaS platform. Scripts are oblivious of the distributed architecture, provided that data is mounted locally to each function, or accessible remotely (e.g., via the HTTP protocol). The programmer can also use the disaggregated in-memory computing layer of Crucial within a script. For instance, the following oneliner synchronizes 100 functions on a barrier named `foo`: `$> seq 1 1 100 | parallel -n0 sshell barrier -n foo await -p 100`. Crucial and the serverless shell are detailed in deliverable 4.3.

Another example of language-level transparency is Fiber [32]. Fiber implements an alternative Python multiprocessing library that works over a scalable Kubernertes cluster. Fiber supports many Python multiprocessing abstractions like `Process`, `Pool`, `Queue`, `Pipe`, and also remote memory in Manager objects. It demonstrates transparency executing unmodified Python applications from the OpenAI Baselines machine learning project. But Fiber does not support transparent disaggregated storage and memory, and it is limited to Python applications using that library.

The Fabric for Deep Learning (FfDL) [33] system moves existing Deep Learning frameworks like PyTorch or TensorFlow to the Cloud on top of cluster technologies like Kubernetes. [33] transparently provides dependability thanks to checkpointing, intercepting storage flows (file system) using optimized storage drivers to cloud object storage, and supporting locality with a gang scheduling algorithm that schedules all components of a job as a group.

FfDL gives us two interesting insights from their authors about the limitations of this system in scalability and transparency. On the one hand, the scaling was observed to be framework dependent so they could not achieve full scaling transparency. On the other hand, they explain that "the service was then increasingly used by data scientists who wanted as much control over their FfDL jobs as with their local machine". Users wanted to download datasets or Python packages from the public Internet, interactively debug models, and stream logs to local scripts in order to monitor the progress of jobs. Many of these requests were not possible due to security limitations and the architecture of the system, which frustrated some of their users.

Another example of transparency in a serverless context, and also a software output of CloudButton project, is FAASM [34]. FAASM is a serverless runtime that uses WebAssembly [35] as its isolaction mechanism. To execute functions in FAASM, they first need to be cross-compiled to WebAssembly (WASM). The WASM language specification allows the cross-compilation step to succeed in the presence of undefined symbols declared as `imports` [36]. The host sytem provides the implementation for these symbols at runtime. This is the technique that WASM uses to implement a set of system interface calls in a platform-independent way using WASI [37].

FAASM leverages this technique to provide a custom host interface for faaslets to chain functions, and share state. In addition, FAASM provides implementations of popular parallel programming models like OpenMP and MPI. This means that serverless applications running on FAASM can execute OpenMP or MPI applications transparently, with an unbounded parallelism, without having to provision any resources in a truly transparent way. This effectively provides functions with transparent distributed shared memory, as long as they use correctly the existing supported APIs. The overheads introduced when providing such support are explored in deliverable 5.3.

4.3 Serverless Python multiprocessing

As we have seen in the sections above, much work has been done both in academia and industry that seek transparency in order to provide a means for application developers to easily scale their code in the Cloud. However, there is a lack of novel transparency approaches that make use of novel state-of-the-art Cloud technologies, such as serverless services like Function as a Service. At the Cloudbutton project, we believe that serverless services, like disaggregated compute (FaaS) and storage (Cloud Object Storage), provide an opportunity for applications to transparently scale in a massive manner, as long as we comply with access transparency.

As explained before, there are many ways towards transparency. A simple one is at the applica-

tion level: if the parallel programming interface used by the application leverages remote disaggregated resources instead of local resources, but maintain the same interface, then we could provide a drop-in library replacement and port local-parallel applications to distributed Cloud instantly with no prior modifications. This objective is ambitious since one cannot just replace local resources with disaggregated resources and expect the same behaviour or performance, since local latencies are much lower compared to network latencies [19].

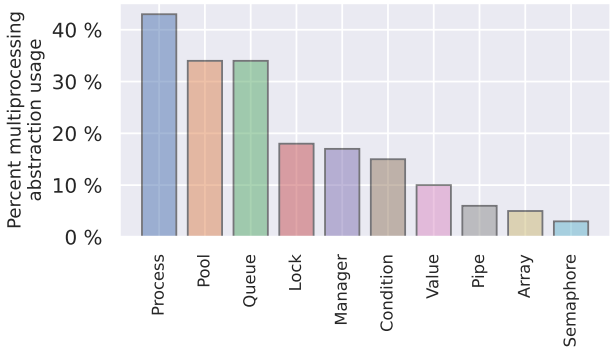


Figure 1: Percentage of usage of the main multiprocessing abstractions of the top 100 most starred GitHub Python repositories that use multiprocessing.

Nevertheless, not all parallel programming models require intensive access to shared memory. On the contrary, many parallel applications just rely on communication and synchronization primitives that could be efficiently disaggregated. In this line, the Python programming language uses processes to achieve true parallelism. Shared state in Python multiprocessing mainly consists of message passing (Pipes, Queues) or remote calls to other processes (Managers). To reinforce this point, we have analyzed the top 100 most starred repositories on GitHub that use Python multiprocessing and found out that Queues and Managers are the most used abstractions for state sharing (Figure 1). In this aspect, the problems and limitations caused by distributed shared memories would not apply, thus transparently adapting multiprocessing Python applications to a distributed environment is more feasible.

In line with Cloudbutton project objectives, we have conducted a performance study [38] to evaluate if the inherent scalability of serverless functions, together with a disaggregated and consistent in-memory storage component, enables to transparently run unmodified Python multiprocessing applications over disaggregated serverless compute resources at scale. We compared the same application with the same workload both in a VM (Virtual Machine) on AWS EC2 and with serverless functions on AWS Lambda, in order to compare execution time, speedup, parallelism and to determine the possible overheads originated by moving to a distributed environment.

For this purpose, we have extended the Lithops serverless computing framework with a module that fully implements the Python multiprocessing interface. This re-implementation leverages serverless functions for processes and Redis database for stateful multiprocessing abstractions (shared state, queues, locks ...). Python applications written with the multiprocessing library can then be transparently ported to the Cloud by only changing the import statement. An overview of the general architecture is depcited in Figure 2.

► **Disaggregated compute resources:** Computation abstractions (like `Process` and `Pool`) leverage the Lithops serverless computing framework. Lithops acts as an abstraction layer on top of AWS Lambda, which enables to execute local serial code over massively parallel serverless functions. A `Process` is mapped to a single Lithops function worker. However, the `Pool` abstraction allows to reuse functions for several consecutive operations on the Pool (like `map` or `apply_async`). A job queue pattern has been implemented where workers are long-lived functions that consume tasks from a queue stored in Redis. Applying jobs onto a `Pool` enqueues tasks to the Redis list instead of invoking new functions. This reduces invocation overhead and prevents stragglers.

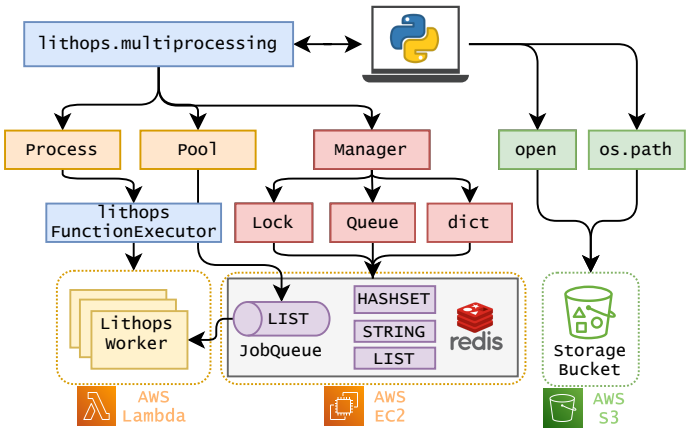


Figure 2: Architecture diagram.

► **Disaggregated memory resources:** Inter-process Communication (IPC) and synchronization abstractions (like `Lock` and `Queue`) are implemented using Redis [39] as a key-value in-memory database. A serverful component is needed to keep the shared state since serverless functions lack addressing and direct communication. Shared objects follow a pattern in which each resource (`Queue`, `Pipe`...) acts as a proxy that access the unique key-value pair in Redis. Redis' different data types and operations facilitate the implementation of stateful abstractions. For example, blocking operations like `Queue.get()` make use of the `LIST` type with the blocking command *BLPOP*.
► **Disaggregated storage resources:** We have also implemented a replica of Python's built-in `open` function and the `os.path` module which allows to transparently read and write files and directories stored on S3 as if it were a local file system. This way, processes can transparently save or recover state to/from storage. However, since object storage works with immutable data, it is not possible to modify or expand (append) a file as it would be done in a traditional network file system without having to rewrite the entire file.

The validation result [38] demonstrates that applications which use stateful abstractions based on message passing, such as queues or pipes, are easy to disaggregate and that the overheads introduced are negligible, obtaining good performance in comparison to the same application running in a big standalone VM. We remark that the port is done by replacing a single line of code, which is a huge deal for scientific application developers that want to massively exploit the parallelism of applications and to further reduce the execution time by increasing parallelism speedup. Access transparency is a key to simplify the whole process of moving applications to the Cloud: legacy applications benefit from access transparency since architecture re-engineering would not be required anymore, and data scientists that are familiar with local-parallel programming can instantly and effortlessly scale their code in the Cloud to process bigger workloads.

The conclusion of our work states that Python's multiprocessing message-passing shared state design is a clear facilitator towards seamlessly porting local-parallel applications over disaggregated serverless resources in the Cloud. Other programming languages that share the same memory address space would be much more difficult to transparently disaggregate at this level. Moreover, performance is severely affected if shared memory abstractions are heavily used, since distributed shared memory will never be as fast as local shared memory. In addition, we require programming interfaces where compute resources (such as processes) or state resources (such as queues) are clearly defined.

In conclusion, access transparency is currently possible with some caveats. However, we are optimistic that network latencies will be reduced, and therefore overheads too, so that access transparency will provide ability to program the Cloud as a parallel Super-Computer, thus hiding the complexities of distributed systems.

4.4 Limits of disaggregation and transparency

Current data center networks already enable disk storage disaggregation [40], where reads from local disk are comparable (10ms) to reads from the network. In contrast, creating a thread in Linux takes about 10 µs, still far from the 15ms/100ms (warm/cold) achieved today in FaaS settings. With that, compute disaggregation is already feasible when job time renders these delays negligible.

Advances in datacenter networking and NVMs have reduced access to networked storage to 1 µs, however this is still an order of magnitude slower than local memory accesses which are in the nanosecond range [22] (100ns), and local cache accesses in the 4ns-30ns range. This means that local memory cannot be neglected, and should be smartly leveraged by memory disaggregation efforts [41]. Existing efforts in memory disaggregation [42, 30, 43, 15] strive to play in the µs range, which can be a limiting factor.

This is directly related to locality and affinity requirements for many stateful applications. The systems community is starting to acknowledge that stateful services need a different programming model and resource management than the stateless ones [25, 6]. Stateful services have very different requirements of coordination, consistency, scalability and fault tolerance, and they need to be addressed differently. Stateful services show the limits of disaggregation versus locality, since in some scenarios locality still matters.

For now, locality still plays a key role in stateful distributed applications. For example: (i) where huge data movements still are a penalty and memory-locality can be still useful to avoid data serialization costs; (ii) where specialized hardware like GPUs must be used [33]; in (iii) some iterative machine-learning algorithms [44]; in (iv) simulators, interactive agents or actors[45].

Finally, another important limitation is scaling transparency, which means that applications can expand in scale without changes to the system structure or the application algorithms. If the local programming model was designed to use a fixed amount of resources, there is no magic way of transparently achieving scalability, not to mention elasticity.

Workloads that do not need elasticity, such as enterprise batch jobs or scientific simulations, can use disaggregated resources the same way as local as they do not need scalability. However, for more user driven and interactive services, such as internal enterprise web applications, simple porting of the executables (sometimes referred as "lift-and-shift") is rarely enough. The unchanged code is not able to take advantage of the elasticity of disaggregated resources and it is expensive to run code that is not used.

4.5 Challenges ahead

Let us review the major challenges to enable transparency for many applications:

- **Granular middleware and locality**: In line with granular computing [25, 22], we require microsecond latencies in existing middleware (compute, storage, memory, communication). In particular, there is a need to handle extremely short instantiation and execution times and more lightweight container technologies. We also require microsecond latencies in disaggregated storage and memory, messaging and collective communication.

  Granular applications are amenable to fine-grained elastic scaling, but this will not provide adequate performance without data locality. Locality and fine-grained resource management may also reduce the current cost of disaggregated resources. Locality is also needed to scale stateful services with different requirements of coordination, concurrency, consistency, distribution, scalability and fault tolerance. But existing FaaS services provide very limited locality/affinity mechanisms and limited networking, precluding inter-function communications. We foresee that next-generation container technologies may enable inter-container communication and provide affinity services for grouping related entities (e.g. gang scheduling [33]).

- **Memory disaggregation and Computational memory**: Disaggregated memory is still an open challenge and there is no available Cloud offering in this line. Many cluster technologies like Apache Spark, Dask, or Apache Ray rely on coupled and difficult-to-scale in-memory storage.

Fast disaggregated memory and storage services [42, 15] can facilitate the elasticity of many cluster technologies [46].

An important problem here is that disaggregated memory services cannot ignore the memory available in existing server-centric nodes in most Cloud providers. One option is to combine both local and remote memory resources efficiently [41]. Another potential solution here is the recent line on computational memory [47] and in-memory computing devices. Compute and memory locality (similar to mammalian brain where memory and processing are deeply interconnected) may considerably enhance computational efficiency.

- **Virtualization** Accessing disaggregated resources in a transparent manner requires a form of lightweight, flexible virtualization that does not currently exist. This virtualization must intercept computation and memory management to provide access to disaggregated resources, and must do so with native-like performance and no input from the programmer. Current serverless platforms use Linux containers and VMs for virtualization [48, 49], which have proven to be too heavyweight for fine-grained scaling, and inappropriate for stateful applications [34, 25, 6, 50]. Software-based virtualization is a more lightweight alternative that is seeing adoption in the serverless context [51, 34], and as a replacement for Docker [52], but is not yet mature enough to transparently support non-trivial existing applications.

  Virtualization necessarily defines an interface between users' code and the underlying system, but the nature of this interface in a transparent disaggregated context is unclear. Exposing a full Linux API makes porting legacy applications easy as shown in LegoOS [27], but requires heavy engineering and introduces historical idiosyncrasies such as `fork` [53]. WASI [37] aims to provide lightweight virtualization with a subset of POSIX-like calls and a custom libc, but can only support a small subset of existing C/C++ applications. Platform-independent runtimes such as GraalVM [54] raise the virtualization layer into the language runtime itself. This affords flexibility in supporting a range of underlying hardware, but is restricted to a a subset of programming languages and applications.

- **Elastic programming models and developer experience:** In some cases, virtualization technologies cannot solve problems like scaling transparency if the code is programmed to use a fixed amount of resources. We then need elastic programming models for local machines that can be used without change when running over Cloud resources. Such elastic models should take care of providing the different transparency types (scaling, failure, replication, location, access) and other aspects of application behavior when it is moved between local and distributed environments. The local executable APIs may need to be expanded to include elastic programming abstractions for processes, memory, and storage.

  To fulfill the vision of disaggregation and transparency it will also be critical to provide tools for developers, enabling them to code both locally and remotely in the same manner with full transparency. Developers will need to be able to use tools to debug, monitor, profile, and if necessary access control planes to optimize their applications for cost and performance.

- **Optimized deployment**: Existing applications are a blackbox for the cloud, but the transition will imply a "compile to the Cloud" process. In this case, the Cloud will have access over applications' life cycle and it will be able to optimize their execution performance and cost. This means that they can perform static analysis to predict resource requirements, dependencies and potential for hardware acceleration. Future Cloud orchestration services will explicitly leverage data dependencies and execution requirements for improving workloads and resource management thanks to machine learning techniques [55, 56]. This compile process will also allow advanced debugging mechanisms for Cloud applications.

  Transparency efforts for different types of applications will require customizable control planes for applications. Such customization will be based on advanced observability and fast orchestration mechanisms relying on standard services and protocols. Monitoring and interception

of the different resources (compute, storage, memory, network) should be available and even integrated into the data center, enabling coordinated actuators at different levels. This can enable the creation of millions of tiny control planes [57] adapted to the different applications and programming models.

We argue that full transparency will be possible soon in the Cloud thanks to low latency resource disaggregation. We foresee that next generation serverless technologies may overcome the limitations exposed by Waldo et al. [19] more than twenty five years ago. In the next years, we will be able to develop programs without taking care of address spaces, while a modern cloud environment will transparently and efficiently execute those on disaggregated resources.

The next frontier for transparency is to go beyond the boundaries of the data center, and seamlessly support heterogeneous resources in the Cloud Continuum (Hybrid/Edge/Federated/Cloud). Another important challenge is to devise elastic parallel programming models for a single machine that can transparently leverage heterogeneous resources in the Cloud Continuum.

5 CloudButton Architecture

5.1 Overall view

The main goal of CloudButton is to simplify Big Data applications thanks to serverless computing. Our recent vision paper entitled *"Serverless End Game: Disaggregation enabling transparency"* [2] is precisely explaining how the disaggregation of computing resources enabled by serverless computing technologies is going to enable almost full transparency.

This vision has long-lasting implications for the development of applications in the Cloud. In particular, it implies that we can move almost unmodified local applications to the Cloud. When our program uses threads and processes, CloudButton transparently runs this code in remote containers (FaaS, Knative), when our program accesses files and directories, CloudButton transparently resorts to Cloud remote storage (Amazon S3, IBM COS, Ceph, etc.), and when our program interacts with memory, CloudButton transparently accesses disaggregated memory (Infinispan, Redis).

This also means that CloudButton aims to minimize the creation of new Cloud APIs. Instead, it relies on the interception of existing programming libraries for accessing remote computing resources. The project is targeting three major software environments: Python, Java, and C++.

- **Python** is now a very popular language for data analysts with popular tools like Python Jupyter Notebooks and libraries like `Numpy`, `Pandas`, or `scikitlearn`;
- **Java** is also heavily used in MapReduce and machine learning environments such as Apache Spark, Hadoop of SMILE ML library, among others;
- Finally, the project focuses on supporting HPC applications typically written in **C/C++** and using popular environments such as MPI and OpenMP.

As we see in Figure 3, CloudButton project presents three main pillars which correspond to the aforementioned Python, Java, and C++ environments. In the three cases, we see how the project is transparently decoupling the underlying disaggregated Cloud resources, namely compute, storage, and memory.

Aside from being built on top of disaggregated resources, the three software pillars of Figure 3 share the CloudButton goal of expressing a wide range of existing data-intensive applications with minimal changes.

In the case of **Python**, IBM and URV are leading efforts around Lithops and the interception of native Python libraries like `multiprocessing` and `concurrent.futures` libraries. Porting an existing Python multi-threaded application that uses `multiprocessing` can be as simple as modifying some import statements. These efforts have demonstrated simplicity by moving very different Python applications and notebooks to the Cloud in Genomics, Metabolomics and GeoSpatial use cases. There are also ongoing efforts to transparently support Python libraries like Scikitlearn on top of our serverless disaggregated software stack. As we explain in WP3 and D3.2 deliverable, Lithops is a popular open source project with an active community and already used in production inside IBM.

In the case of **Java**, IMT and URV are leading efforts around the Crucial Java Serverless Toolkit. Crucial is also intercepting the standard Java Concurrency APIs, by providing a Serverless Executor that can run threads over Serverless Functions. Crucial also provides shared mutable state and synchronisation primitives in the Concurrency API (`Barrier`, `AtomicLong`, `AtomicList`) over RedHat Infinispan In-Memory middleware. Thanks to these APIs, we demonstrated different machine learning algorithms and applications like KMeans, Logistic Regression, and Random Forests in SMILE, a Java ML library [58]. As we explain in WP4 and D4.2 deliverable, Crucial is an open source project that leverages and benefits from Redhat Infinispan's Java open source community.

In the case of **C/C++** and HPC, IMP is leading efforts around FaasM WebAssembly serverless Toolkit. WebAssemby is a popular Intermediate Language that allows the compilation/execution of multiple programming languages like C/C++, Javascript and many others. FaasM is intercepting POSIX APIs in WebAssembly to execute threads in serverless containers that can also benefit from locality and shared memory. IMP is demonstrating that is is possible to port existing C++ HPC

applications using MPI and OpenMP. As we explain in WP5 and D5.2 deliverable, FaasM is an open source project that has raised the interest of the WebAssembly and serverless community.

In summary, all three pillars software outcomes are committed to the global objective of providing simple programming models for serverless cloud infrastructures targeted to existing data-intensive applications and implying as few changes as possible.

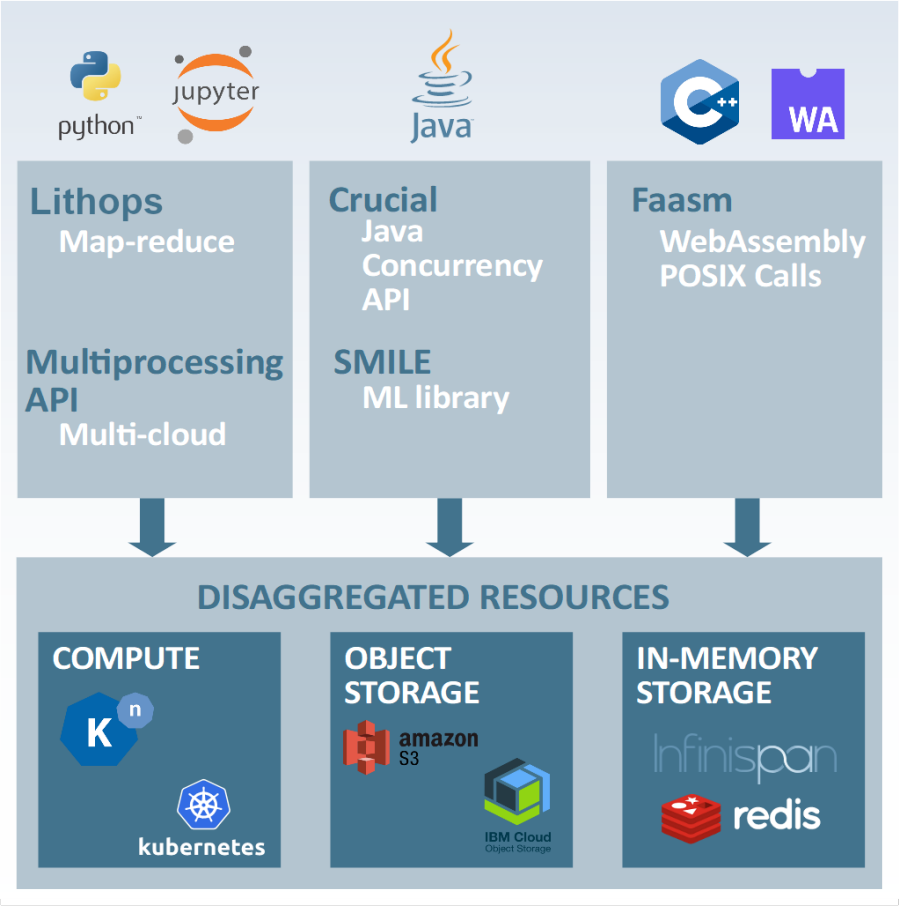


Figure 3: Big picture of the CloudButton architecture

5.1.1 Integration among the different components and contributions

As the project advances, we have seen an increase on the maturity and popularity of Knative and CNCF technologies as standard serverless building blocks for Kubernetes. For this reason, we believe that Knative can be the unifying framework for all the software components created in the context of the CloudButton project. Moreover, the adoption of Knative is a step towards serverless standardization: it simplifies the building of container-based, serverless applications that can deploy and run on any cloud: public, private, and hybrid.

On the one hand, CNCF K8s is becoming a Cloud standard, with many public Cloud providers supporting those technologies. But Knative as a serverless layer over K8s has a strong potential for **portability** and standardisation. Google is now offering a multi-tenant serverlss K8s and Knative in their Google Cloud Run service, but many others are following like IBM Code Engine, VMware/Dell, NetApp, or Alibaba among others.

The three main pillars of the project run on top of CNCF technologies in a transparent way. For instance, Lithops has support for running over Knative on any Kubernetes cluster [59], and Faasm is already integrated with Knative [60].

ATOS's efforts are focused on instrumentation and SLAs of Knative clusters using K8s technologies. This means that all their work is compatible with the different software stacks built on top of it.

Also, the testbed provided by ATOS is built around Knative tachnologies.

RHT Infinispan has actively worked in the integration of Infinispan with K8s technologies, first providing a K8s operator, and then providing an auto-scaling service that can be integrated in a Knative cluster. Redhat also improved the interoperability of Infinispan with other programming languages. Again, all pillars can benefit from an Infinispan in-memory service deployed in the Knative cluster and offering auto-scaling support.

Finally, an essential service for interconnecting and integrating different services is Serverless Orchestration relying on Knative and CNCF CloudEvents standards. We refer here to WP3 D3.2 section on Serverless Orchestration, and in particular to the recent publication title "TriggerFlow: Trigger-based Orchestration of Serverless Workflows" [61]. In this work, we present an integrated event-based architecture that enables the orchestration of Cloud technologies.

IBM and URV are working together in the integraton of serverless orchestration technologies using Knative Eventing, KEDA, but also serverless orchestrators for K8s like Argo and KubeFlow. In the end of the project, we will be able to orchestrate Big Data pipelines with stages using different technologies like Python, Java or FaasM.

The convergence towards Knative will enable the orchestration of serverless workflows using different components of the CloudButton architecture. In order to allow these workflows orchestration, each component will provide an invocation endpoint and termination events. Platforms like ARGO and Triggerflow (developed in the context of this project) will be used to put together workflow steps implemented with different serverless technologies and even other steps that require serverful components.

5.2 Software Releases

In this section, we will describe the different software projects that make up the main software components of the CloudButton Architecture. The unifying framework for all components is the serverless cluster using CNCF K8s technologies.

- Serverless Infrastructure (OpenWhisk, Knative, Prometheus): IBM and ATOS have integrated SLA monitoring components in Kubernetes in order to provide fault-tolerance, and QoS support to Serverless Data Analytics pipelines.

- Serverless Orchestration (Airflow, Kubeflow, Argo): IBM and URV have created tools enabling the orchestration of Big Data pipelines over serverless functions and containers. The tools (TriggerFlow, Argo, Aitflow plugin) provide declarative DAGs (Directed Acyclic Graphs) for the definition of the pipelines. Such DAGs are leveraged by the underlying Serverless Infrastructure to optimize resource usage. For example, we have extended Apache Airflow with new FaaS Big Data Operators [62] and we also have adapted Argo for Big Data pipelines on serverless containers.

- Mutable State Middleware (Crucial, Infinispan): IMT, URV, and RHAT have created a novel disaggregated mutable middleware including consistent data structures and programming abstractions for stateful Big Data analytics over Infinispan. We offer proof of concept machine learning algorithms packaged as cloud functions that can be executed and orchestrated by CloudButton Core in a K8s cluster. RHAT has also integrated Infinispan and this middleware in the K8s stack with a K8s operator, and it provides controllers for flexible auto-scaling of ephemeral and replicated Infinispan clusters. For an in-depth description of the mutable state middleware see D4.2 [63]. Source code repositories: `https://github.com/danielBCN/crucial-dso` / `https://github.com/infinispan/infinispan`.

- WebAssembly FaaS Runtime (Faasm): Imperial have created a novel lightweight and polyglot FaaS runtime over WebAssembly technology. This middleware offers code-shipping models where lightweight functions can be colocated and access local shared memory in an efficient way. Faasm is described in detail in D5.2 [64]. Source code repository: `https://github.com/lsds/faasm`.

- Lithops toolkit multiprocessing API: a multicloud framework that enables the transparent execution of unmodified, regular Python code against disaggregated cloud resources. It provides the same API as Python's standard `multiprocessing` and `concurrent.futures` libraries. Source code repository: `https://github.com/cloudbutton/cloudbutton`.

5.2.1 Multi-cloud support

Current serverless computing platforms range from the traditional Functions-as-a-Service [65, 66] to the newest Container-as-a-Service [67, 68] services. Moreover, tens of open-source frameworks backed by Kubernetes are available in the community [69, 70]. With such an array of options, it is not unreasonable for a user to wonder: "*Which serverless computing service should I use?*", particularly under the shadow of the vendor lock-in problem. Vendor-specific APIs and SDKs make it difficult for users to move their applications from one underperforming cloud to another, simply because switching to a better provider is deemed too costly. In this sense, the continuous evolution of serverless APIs and features does nothing but to further aggravate this problem. In such a scenario, a multi-cloud approach can bring the needed flexibility to leverage the best of each cloud platform and overcome vendor lock-in, so developers can fully enjoy the benefits of serverless computing.

Therefore, the CloudButton toolkit adopts a multicloud-agnostic architecture, in an effort to ensure portability and overcome vendor lock-in problems. The CloudButton toolkit enables transparent access for users to virtually unbounded multicloud resources as nothing more than writing a program with a familiar language.

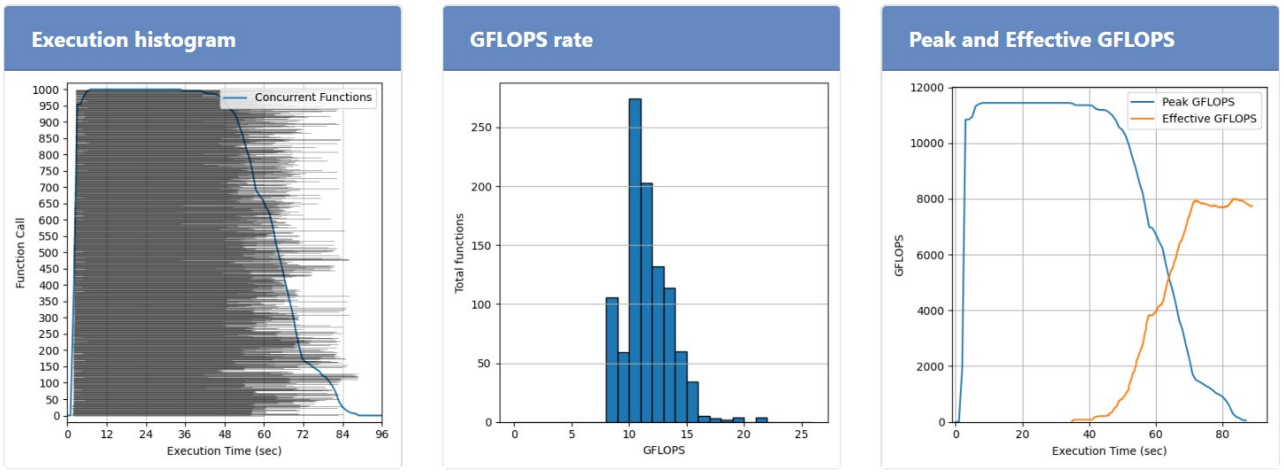


Figure 4: CloudButton Serverless Benchmark – IBM Cloud Functions results

An interesting example of the multi-cloud support is the CloudButton Serverless Benchmark (`https://github.com/lithops-cloud/applications/tree/master/benchmarks`), a helpful tool to compare serverless offerings (in terms of compute and storage) across different public cloud providers. The benchmark evaluates compute power and scalability of cloud providers by running multiple compute intensive tasks concurrently. It also evaluates the throughput of read and write operations to the object storage services of different cloud providers. As of today, the benchmark supports 6 FaaS services (IBM Cloud Functions, AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, Google Cloud Run, and Alibaba Aliyun Function Compute) and 5 object storage services (IBM COS, AWS S3, Microsoft Azure Blob, Google Storage, Alibaba Aliyun Object Storage Service).

6 Metabolomics use case

In the current reporting period, we have achieved all the planned objectives.

6.1 Description of the use case

Spatial metabolomics is a field of omics research focused on the detection and interpretation of metabolites, lipids, drugs, and other small molecules in the spatial context of cells, tissues, organs, and organisms. Spatial metabolomics is a rapidly emerging field, fueled by the strong and ever-growing need in biology and medicine to characterize biological phenomena in situ, as well as by the recently revealed key roles of metabolism in health and disease. This field is concerned with a variety of biomedical questions, including the tumor molecular microenvironment, functions of immune cells during homeostasis and immunotherapy, interactions between host and microbiota and their contribution to inflammation, regulation of early development, metabolic regulation of epigenetics, and metabolic dysregulations during infection and inflammation. Over the past decade, this growing interest has stimulated rapid progress in the development of enabling technologies – in particular, imaging mass spectrometry (MS) – that have achieved unprecedented sensitivity, coverage, and robustness as they have become accessible to biologists (Figure 5).

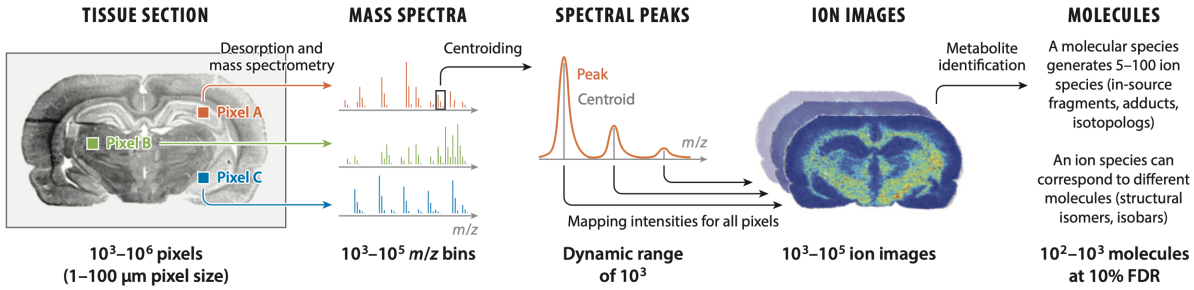


Figure 5: An imaging mass spectrometry (MS) dataset represents a collection of spectra acquired from a raster of pixels representing the surface of a tissue section. An ion image represents relative intensities of the ion across all pixels. An imaging MS dataset can represent spatial localization of up to 10^3 molecules.

We have developed METASPACE, a global community platform for spatial metabolomics populated by a large community of users. The cornerstone of METASPACE is a computational engine for metabolite annotation which searches for metabolites, lipids, and other small molecules in an imaging MS dataset. The engine estimates the False Discovery Rate (FDR) of metabolite annotations that provides quality control and – as demonstrated in other -omics – makes annotated spatial metabolomes comparable between datasets, experiments, and laboratories. We created a user-friendly web app for data submission and for interactive exploration of annotated metabolite images. By sharing their results publicly, METASPACE users cooperatively created and continuously populate a knowledge base of spatial metabolomes.

Earlier in the Horizon2020 project METASPACE (2015-2018), was open-source cloud software METASPACE. METASPACE integrates a high-performance cloud computing engine, a webapp for data submission, results search, browsing, analysis, and sharing, as well as a knowledgebase of private and public datasets and results from them. Since 2017, METASPACE became a major tool in spatial metabolomics with over 1000 registered users from over 100 labs, with many using it every day (Figure 6). We processed almost 20K submissions, with over 1K submissions per month lately. Importantly, 30% of these submissions were shared publicly. This represents the largest public data collection in spatial metabolomics (and one of the largest in metabolomics in general) and, with provided metadata, a continuously-populated knowledgebase of spatial metabolomes.

Importantly, METASPACE requires scalable computing, taking into account the growth of the field (Figure 7) as well as the diversity of the datasets submitted to METASPACE in its nature and

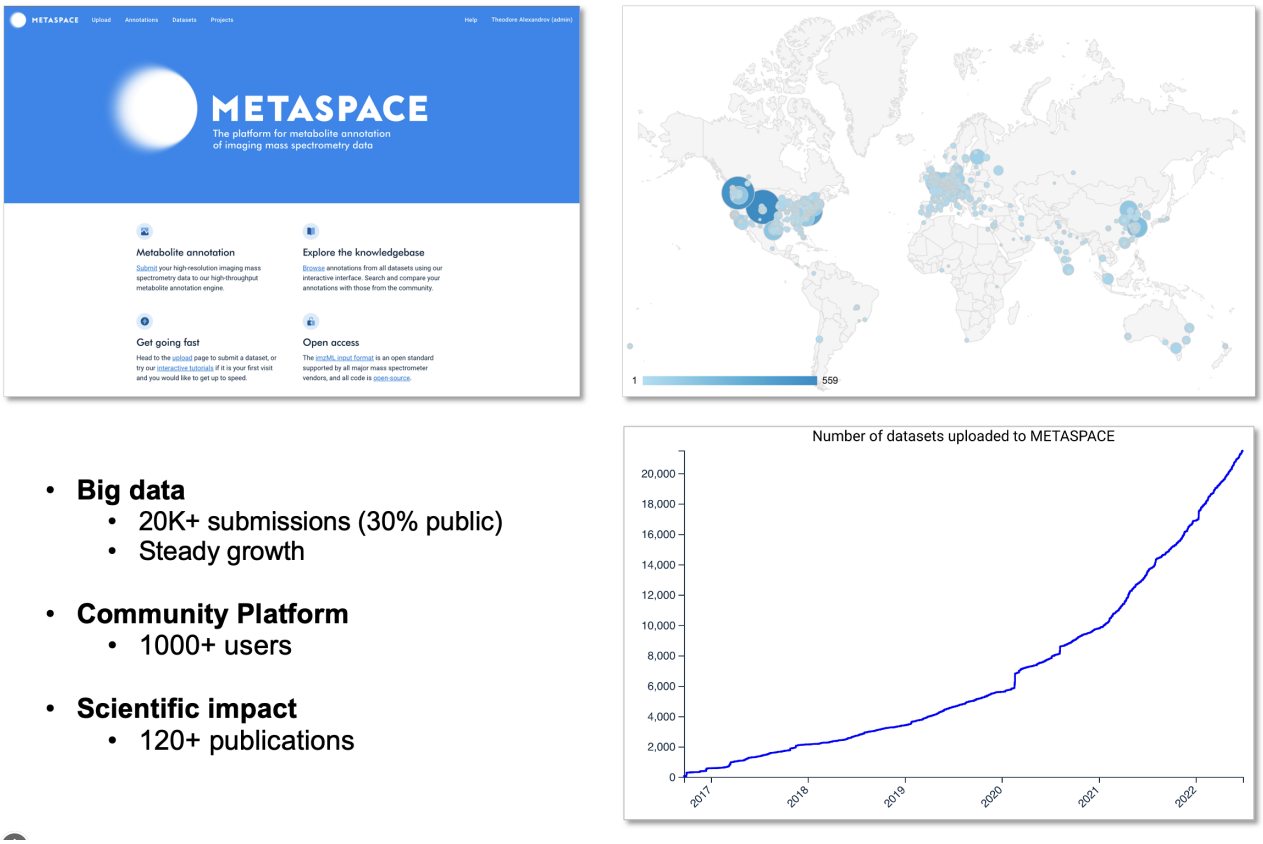


Figure 6: The METASPACE cloud platform (`http://metaspace2020.eu`), the geographical map of its worldwide usage, the growing numbers of submissions and the key points characterizing the big data aspects, its growth, community support, and scientific impact.

size (Figure 8).

6.2 METASPACE and the Big Data challenge

METASPACE is a big data cloud platform used by scientists from all over the globe to submit multi-gigabyte datasets generated in various experiments by various instruments, often with a high uncertainty in the quality or content of the data. The critical aspect in its operation is that the data is sometimes too big to load entirely into memory. Every pixel in the spatial metabolomics image can be considered as a data point containing thousands of molecules, with the number of pixels reaching as high as a million. This generates huge amounts of data (sometimes larger than 1TB) that can't be loaded entirely into memory but still needs to be sorted, chunked, annotated, and collected together. The challenges this presents are how to process data in concurrent tasks running on different machines and how to assemble the results and represents an example of the big data analysis. Obviously, we don't have the luxury of having a super machine with unlimited memory and computing power to load all data into memory and process it there. But even if we did, it would still be challenging to efficiently utilize the computing resources of this super machine since developers would need to take care of running multiple threads, processes, etc. in coordination. The METASPACE knowledgebase has two components: raw imaging mass spectrometry data and metabolite images produced from the raw data by our bioinformatics engine. Each raw dataset is a multi-gigabyte hyperspectral image of over 1 million of channels. Metabolite images are obtained from the raw data with the help of advanced custom bioinformatics and data-intensive computing currently powered by the Apache Spark technology. However, even this implementation is not scalable enough to deal with the big data of ever-growing and diverse spatial metabolomics datasets, in particular due to the rising popularity of our platform. In addition, the results represent a fraction of percent of the raw

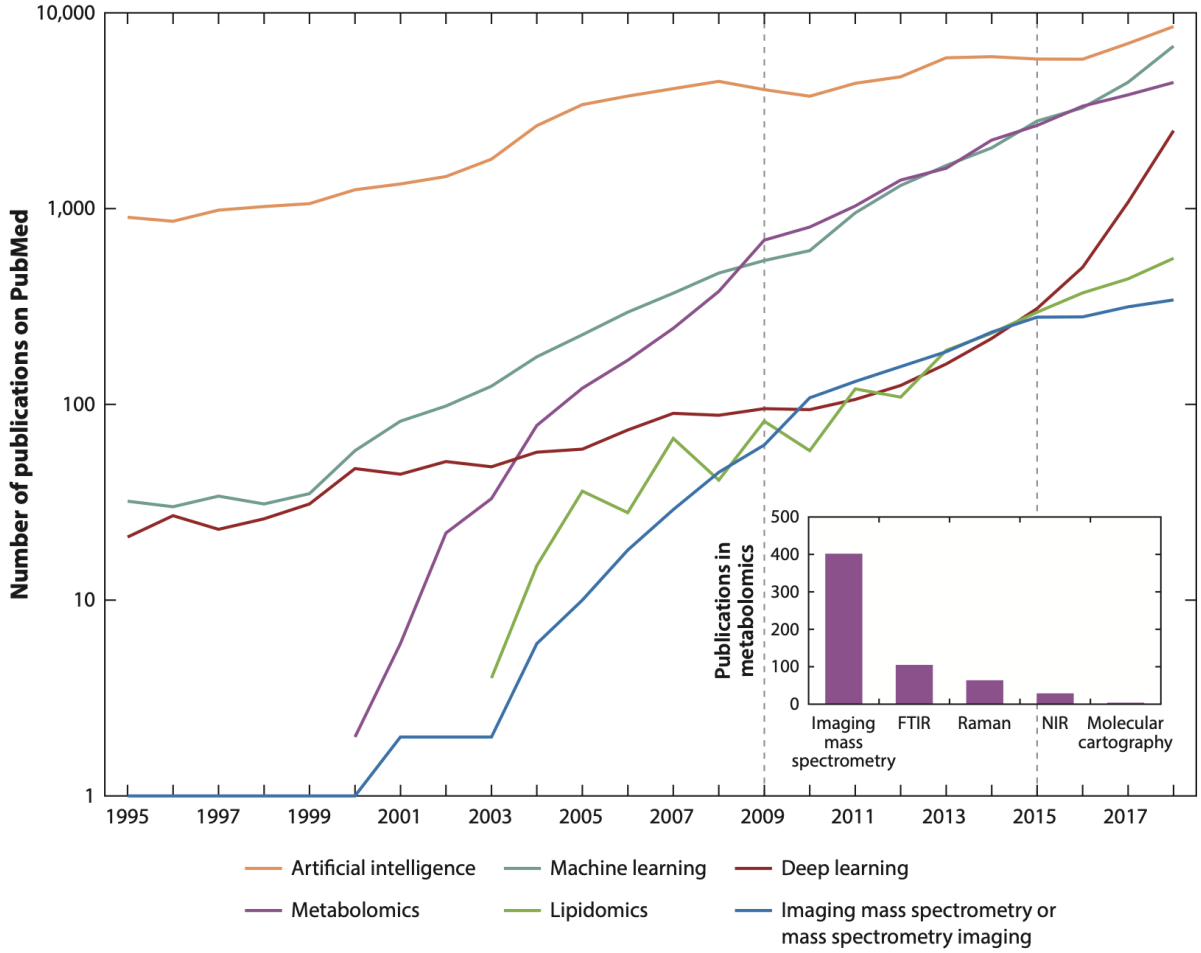


Figure 7: The popularity of different technologies in the life sciences and biomedicine and their evolution over time. The plot shows the numbers of PubMed-indexed publications in a given year containing the keywords shown in the figure key. We highlight three time periods, before 2009, from 2009 until 2015, and after 2015, which we discuss in the main text. The inset shows the popularity of several technologies for metabolomics applications from 1995 until 2018.

data and are accessed either through a web-app for interactive exploration or through a GraphQL API particularly by using Python Jupyter notebooks. However, currently most of the raw data is left unexplored, often called as "black matter" in spatial metabolomics. The main challenge of using Apache Spark is the need to hard-code or predict in advance the resources needed at every point in time of processing the data. This is exactly the deficiency that can be addressed by the serverless processing paradigm.

6.3 METASPACE and CloudButton

Serverless paradigm is a very attractive approach to resolve challenges of METASPACE. It can almost instantaneously allocate large amounts computing resources and users only pay for these actually used resources. However, there is still the remaining challenge of how to effectively scale the METASPACE engine for serverless processing of a dataset stored in the cloud object storage, how to monitor the executions, and execute all tasks as a single “logical” job in IBM Cloud Functions. It also a challenge how to decide on right parallelism and resources required per each annotation job. And obviously we don’t want to rewrite METASPACE from scratch to leverage serverless, rather integrate serverless into METASPACE by using push to the cloud approach. To address the challenges

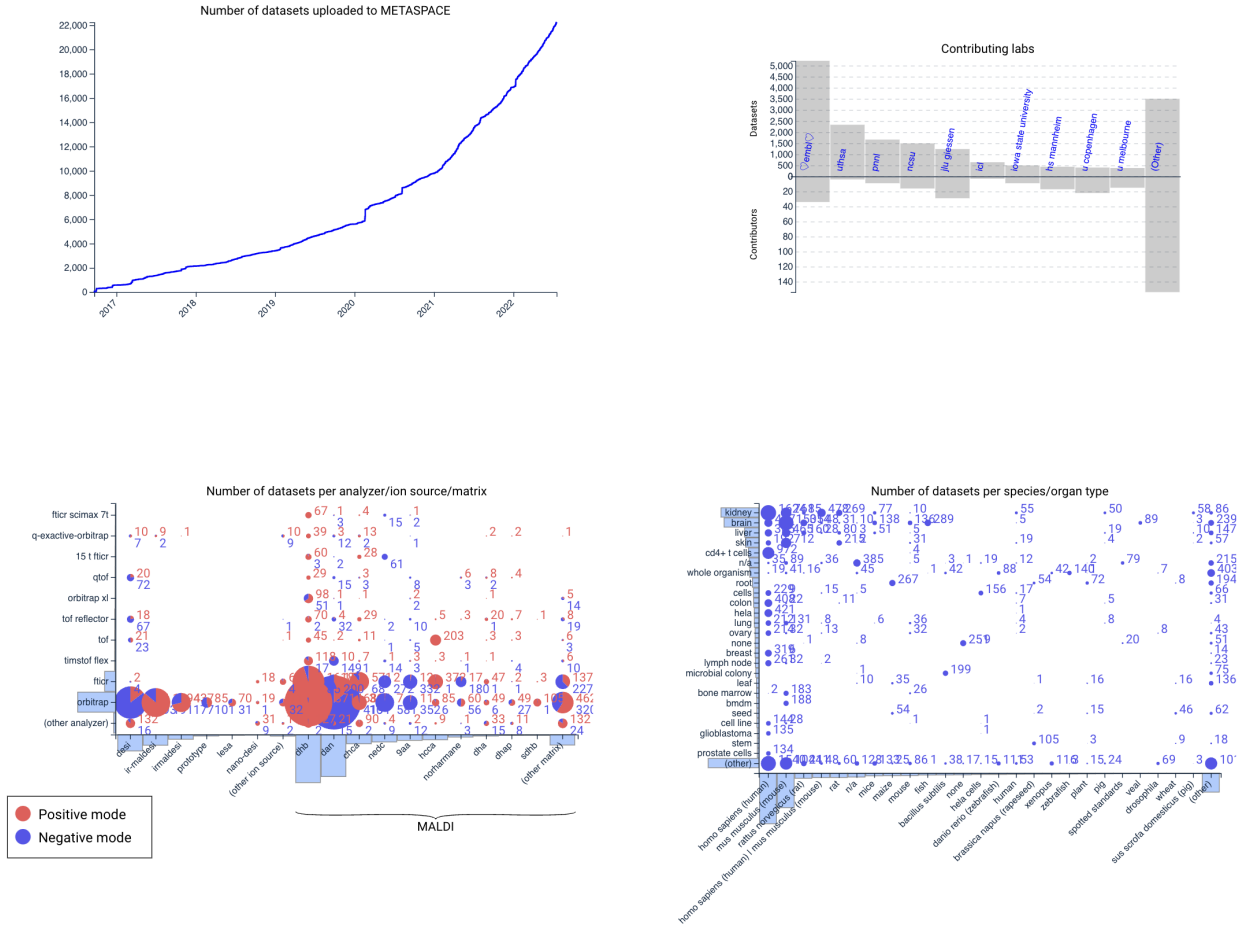


Figure 8: The diversity of the big data in METASPACE covering multiple submitting labs, datasets from various technological platforms, and biological sourses of the data.

above, this is exactly what we addressed in the Lithops framework developed in the context of the European Horizon2020 project CloudButton. Together with IBM Research, we have developed our solution based on the open source Lithops framework. Lithops introduces serverless computing with a minimal effort and brings automated scalability for massive data processing. The goal of Lithops is to enable an easy move to serverless by providing a "push to the cloud" experience: Users can focus on their code, while Lithops focuses on the code execution in the cloud.

6.4 METASPACE-Lithops - The first step to Serverless

EMBL and IBM together have developed a serverless implementation of the METASPACE. The implementation was deployed over IBM Cloud and is available at our GitHub `https://github.com/metaspace2020/Lithops-METASPACE` with 288 commits as of July 2022 (Figure 11): The figure Figure 16 shows a high-level approach of our design. Our core approach is decentralized and completely serverless, where we let the Lithops framework determine the appropriate scale of parallelism needed to process input datasets. To achieve this, our code evaluates the input datasets and then decides on the number of serverless actions required, with the aim to maximize performance and the costs of the processing.

This approach allows us to dynamically adjust the amount of compute resources while the data is being processed—which is in contrast to a Spark-based approach, where the amount of compute resources is determined before starting the data processing and can't be adjusted as the processing progresses.

Figure 12 demonstrates the key aspects of using an early implementation of the serverless version

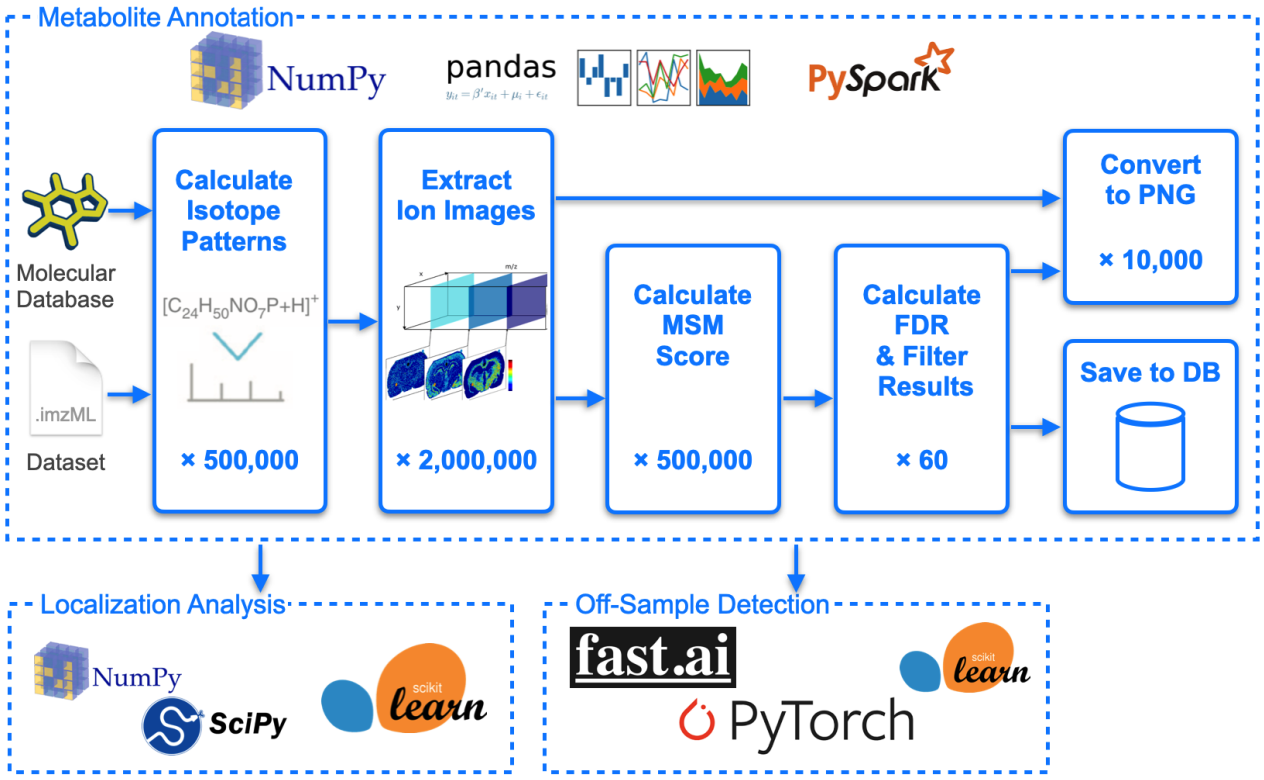


Figure 9: The workflow of the serverful METASPACE (existed before this project) indicating the steps and the numbers of invocations of every step for processing of one dataset).

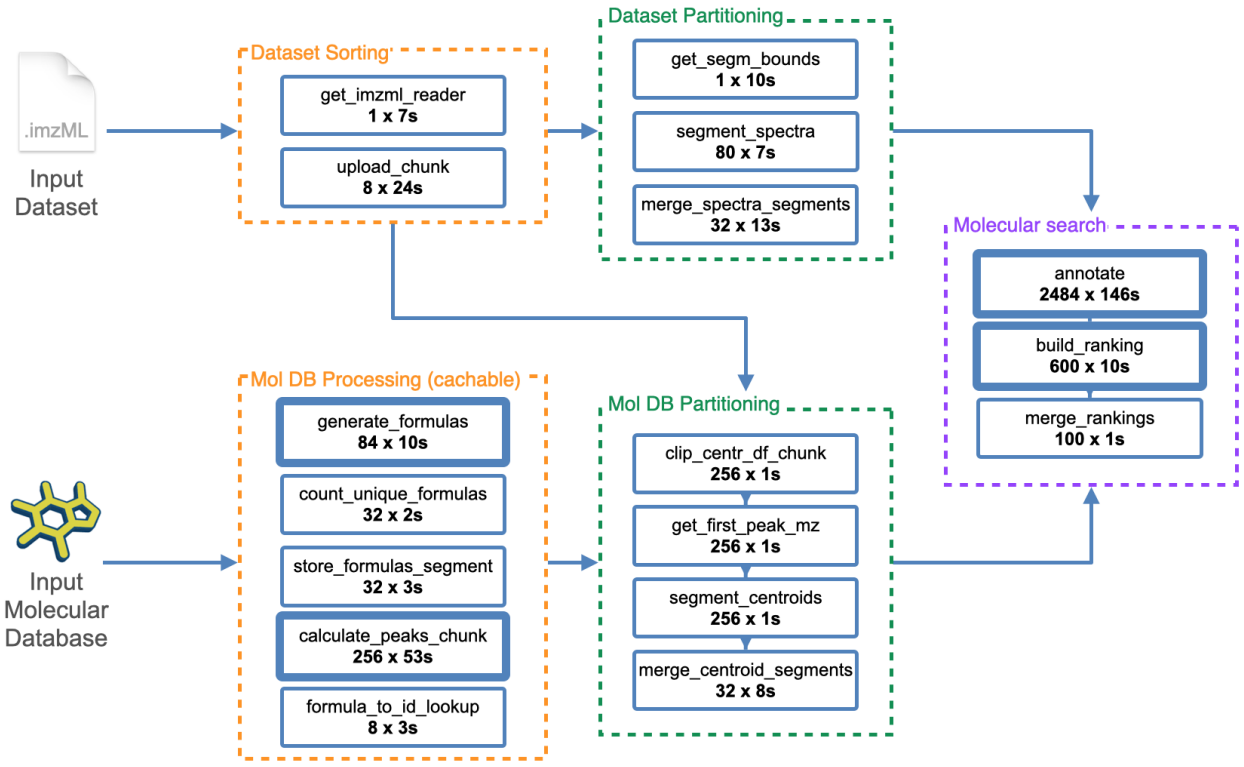


Figure 10: The architecture of serverless METASPACE

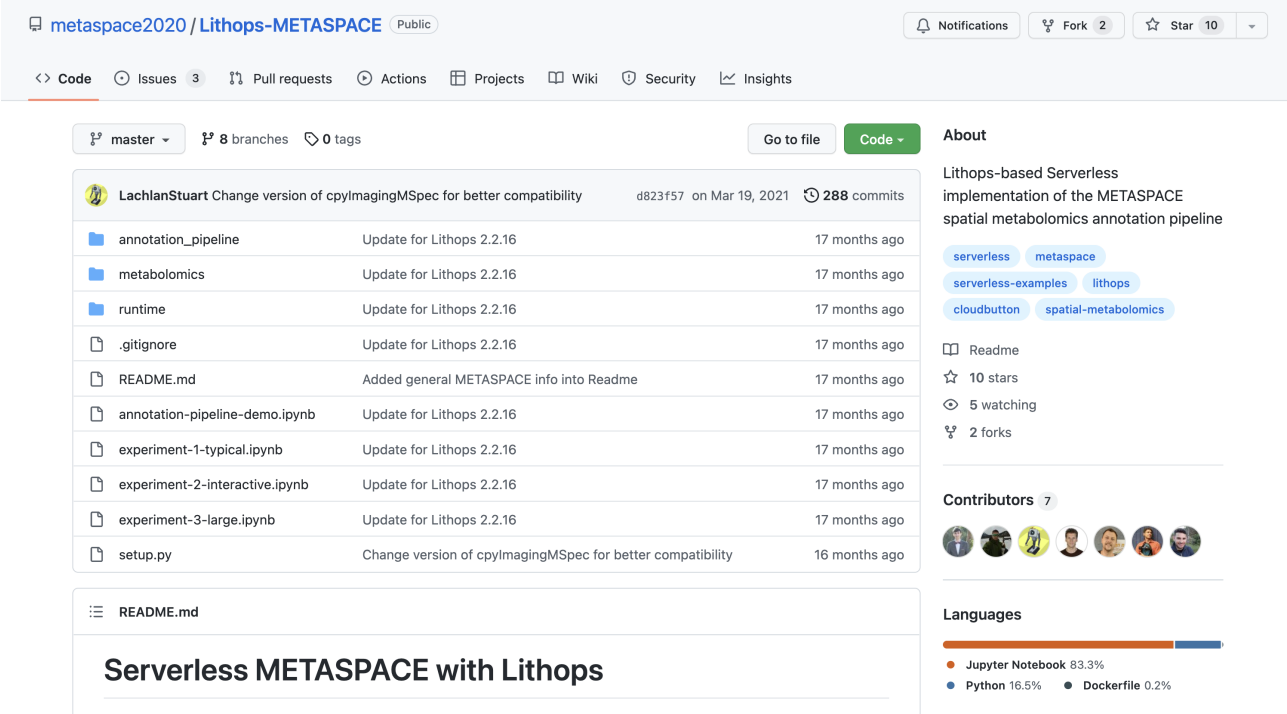


Figure 11: The GitHub repository of the Lithops version of METASPACE which is deployed in production since March 2021.

of METASPACE, implemented in PyWren (early version of Lithops).

6.5 Experiments

EMBL has prepared three experiments representing the main computing scenarios in METASPACE. **Experiment 1, "Typical use case"**, is representative of a normal use-case on METASPACE, which makes it suitable for head-to-head comparisons. There is often limited time available on the higher-spec PC used for initial data capture as it is a shared resource, so usually the analysis will be performed from scientists' or students' lower-spec laptops. **Experiment 2, "Interactive reprocessing"**, is representative of a new type of functionality that we currently don't support in METASPACE because it's uneconomical with the serverful approach. While looking for specific compounds, scientists tend to have relatively short lists of molecules of interest, and iteratively try different adducts or modifiers until they find the data they're interested in. **Experiment 3, "Stress test"**, aims to ensure that the limits of are similar to METASPACE, this is one of the larger datasets that has been processed. For every experiment, we have prepared the relevant datasets and databases as well as prepared the metrics to be used for benchmarking.

6.6 Benchmarking datasets and metrics

EMBL has prepared the benchmark datasets and shared them with the partners as well as publicly through the specially-setup repository: https://github.com/metaspace2020/Lithops-METASPACE#example-datasets. This included datasets from six tissue sections provided by EMBL and our collaborators, as well as 12 molecular databases which are used in METASPACE for molecular annotation. In every experiment, molecules from a database are searched for in a dataset. The datasets and databases were selected so that their combinations represent various scenarios: from small to typical to large to huge computing scenarios.

The datasets sizes range from 0.05 GB to 56.7 GB (see details at https://github.com/metaspace2020/Lithops-METASPACE/tree/master/metabolomics#dataset-configurations) that is representative for the highly variable sizes of datasets processed by METASPACE.

EMBL has formulated the following metrics to be used for benchmarking:

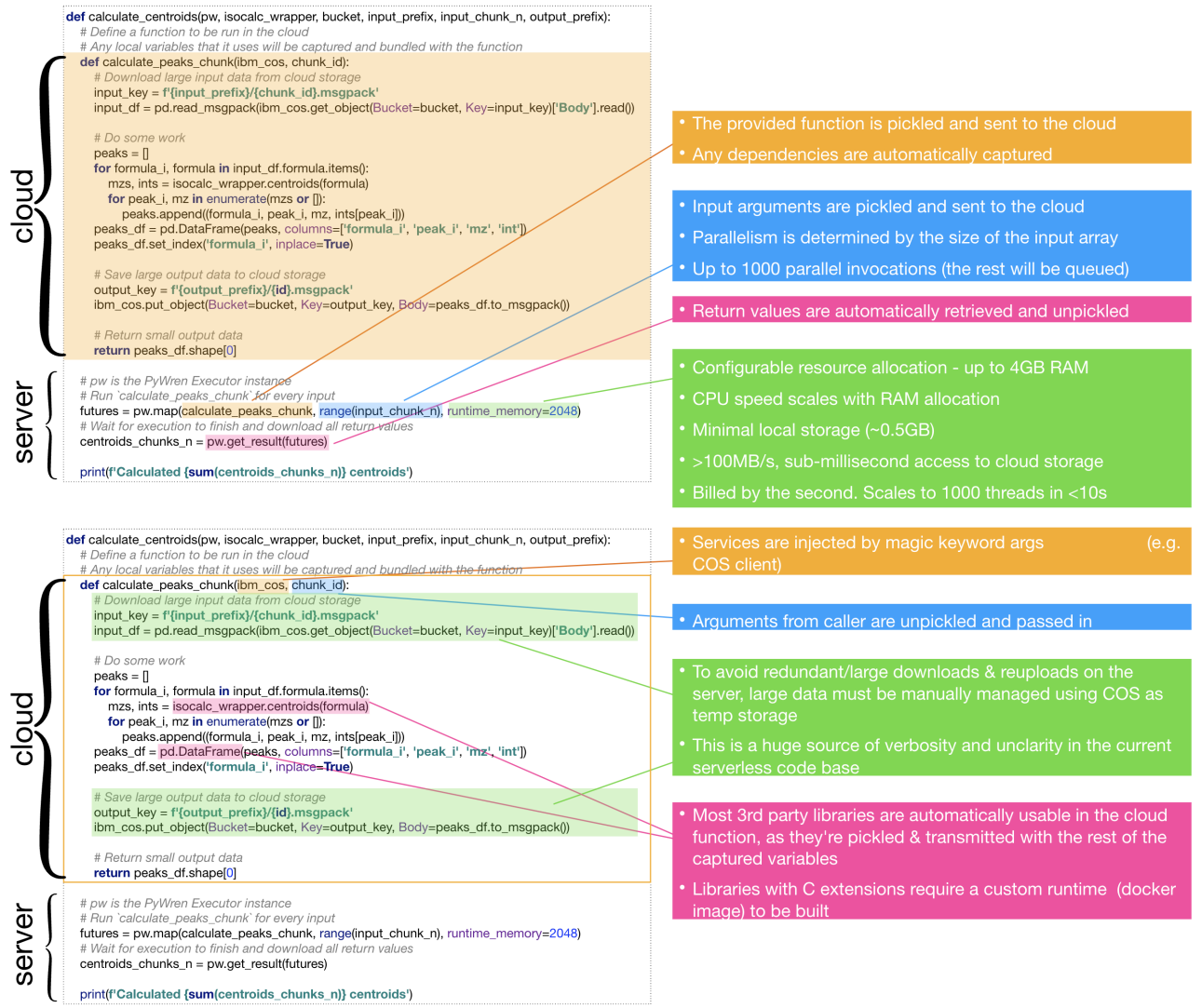
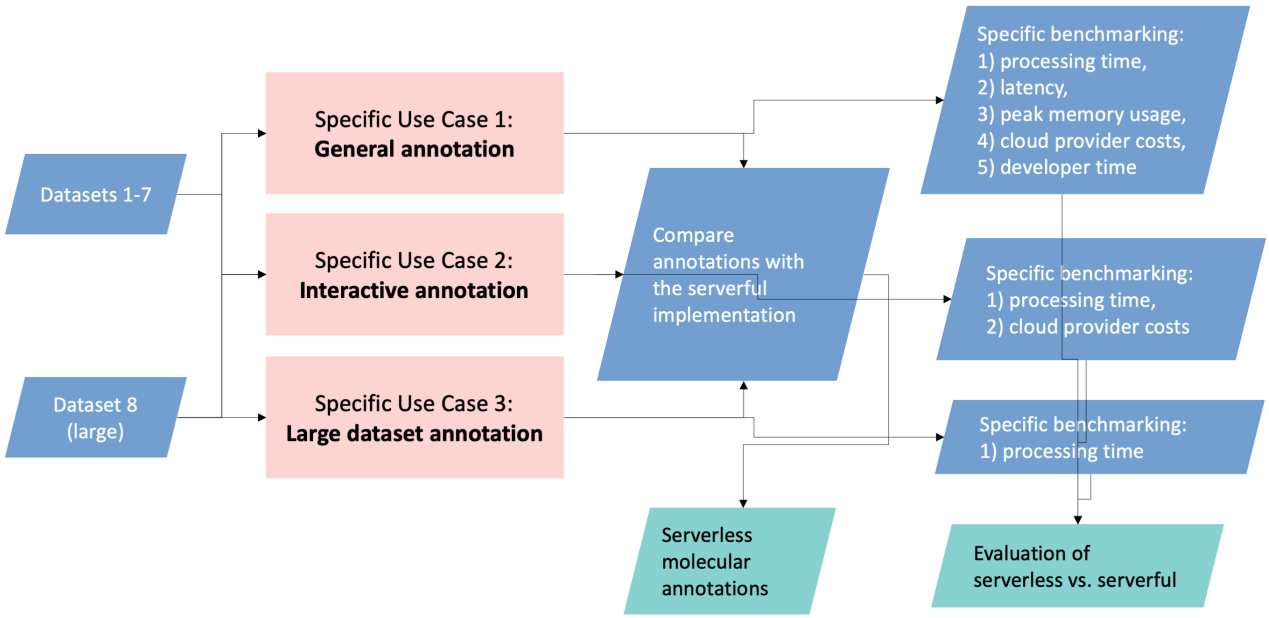


Figure 12: Demonstration of the key aspects of using an early version of serverless METASPACE.

- total processing time
- cloud provider costs, and finally
- developer time

For every experiment, we decided which metrics are critical and which goals should be achieved (Figure 13).

After performing initial evaluation benchmark, we have identified 3 bottlenecks which were responsible for 80%-90% of processing time which are at the same time “embarrassingly parallelizable” (Figure 15). Moreover, we have identified 2 bottlenecks (the partitioning steps) which are responsible for most of the development pain (Figure. We also concluded while Serverless is very attractive solution, some of the legacy code performs better when leverage large memory, which is not available for FaaS solutions. To make METASPACE most efficient, while enabling it to run some of the legacy code with large memory and some code distrbute as a serverless invocation, we had come with hybrid approach which appears to be the most effective for METASPACE.



Example notebooks

The main notebook is `pywren-annotation-pipeline-demo.ipynb`, which allows you to run through the whole pipeline, and see the results at each step.

There are also 3 notebooks prepared for benchmarking that can be run with Jupyter Notebook:

1. `experiment-1-typical.ipynb` - Demonstrates running through the whole Serverless metabolite annotation pipeline with a typical dataset,
   downloading the results and comparing them against the Serverful implementation of METASPACE.
2. `experiment-2-interactive.ipynb` - An example of running the pipeline against a smaller set of molecules, to demonstrate the potential of Serverless to provide low-latency access to computating resources.
3. `experiment-3-large.ipynb` - A stress test that runs the Serverless metabolite annotation pipeline with a large dataset and many molecular databases.

Figure 13: The specific use cases formulated for benchmarking of the serverless version of METASPACE, the decision diagram for evaluating the improvements compared to the state-of-the-art serverful version, as well as a screenshot from our GitHub repository where we have prepared Jupyter notebooks for every specific use case `https://github.com/metaspace2020/pywren-annotation-pipeline#example-notebooks`.

6.7 METASPACE-Lithops - The hybrid solution

We have designed a new architecture that would combine serverless and the use of large virtual machines (VMs); see Figure 16.

In this unique approach, Lithops creates a powerful VM, deploy METASPACE sorting code into this VM, perform the execution. Lithops monitors the VM, execution progress and upon completion, Lithops will persist the results into IBM Cloud Object Storage. Then Lithops will automatically delete the VM once all results were persisted in the object storage. The next phase, Lithops will use function as a service with parallel invocations as required, that will process dataset generated from the previous stage. From user point of view, this is pure cloud button experience, since all happens transparently to the user without him manually to create VMs, monitor execution, handle data transfer, and so on. The hybrid implementation combining serverless and VMs is used in the production version of METASPACE since March 2021.

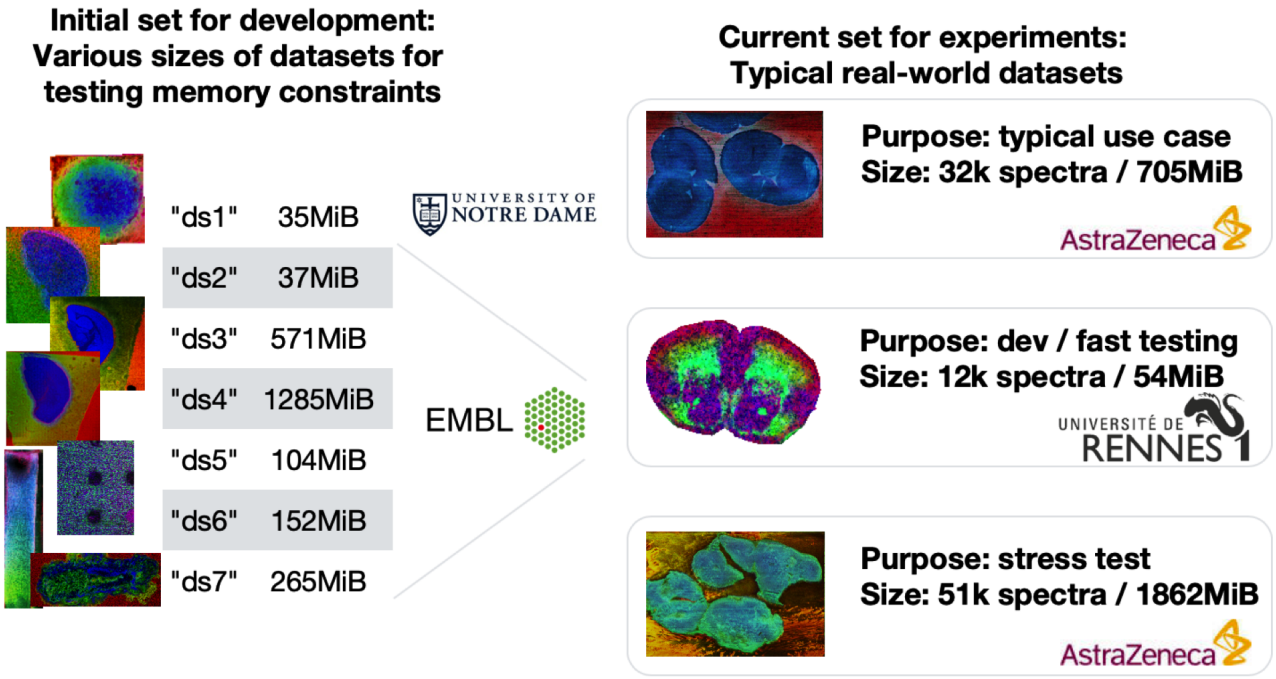


Figure 14: Overview of the datasets provided by EMBL and its collaborators for benchmarking the serverless implementation of METASPACE. With owners permission, we have made these datasets public through our GitHub `https://github.com/metaspace2020/pywren-annotation-pipeline`.

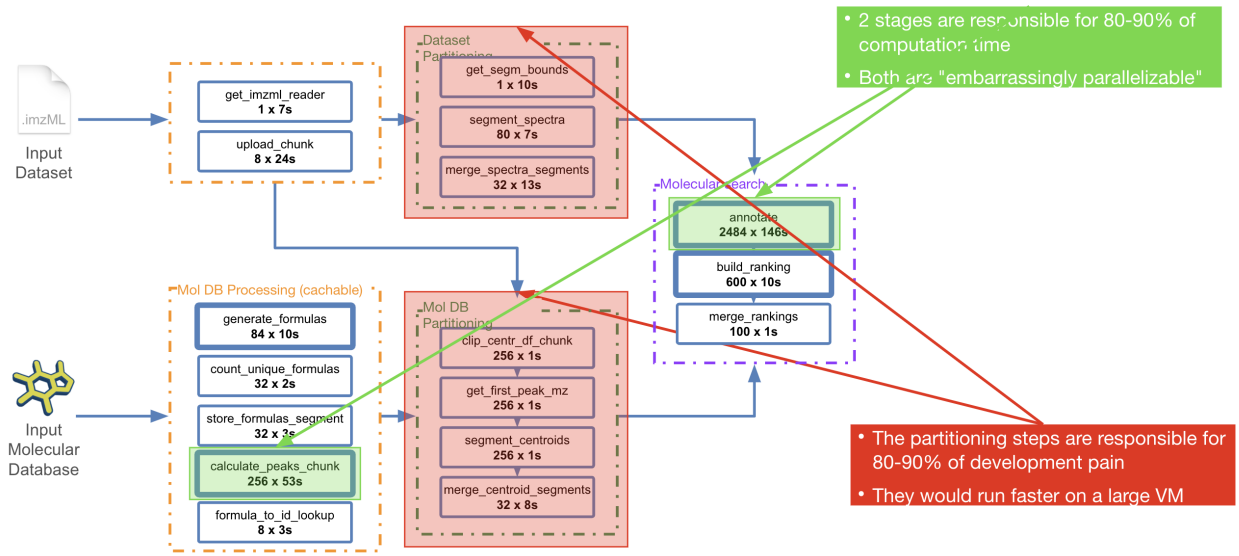


Figure 15: Analysis of the serverless implementation of METASPACE. We have identified bottlenecks which are responsible for the majority of computational time and "developer pain" that led us to developing a hybrid solution (see next section).

6.8 Benchmarking results, KPIs

For the benchmarking, EMBL and IBM have prepared Jupyter notebooks which use the serverless implementation of METASPACE: `https://github.com/metaspace2020/Lithops-METASPACE#running-the-exampl`
Each notebook executes the serverless METASPACE and performs the benchmarking according to

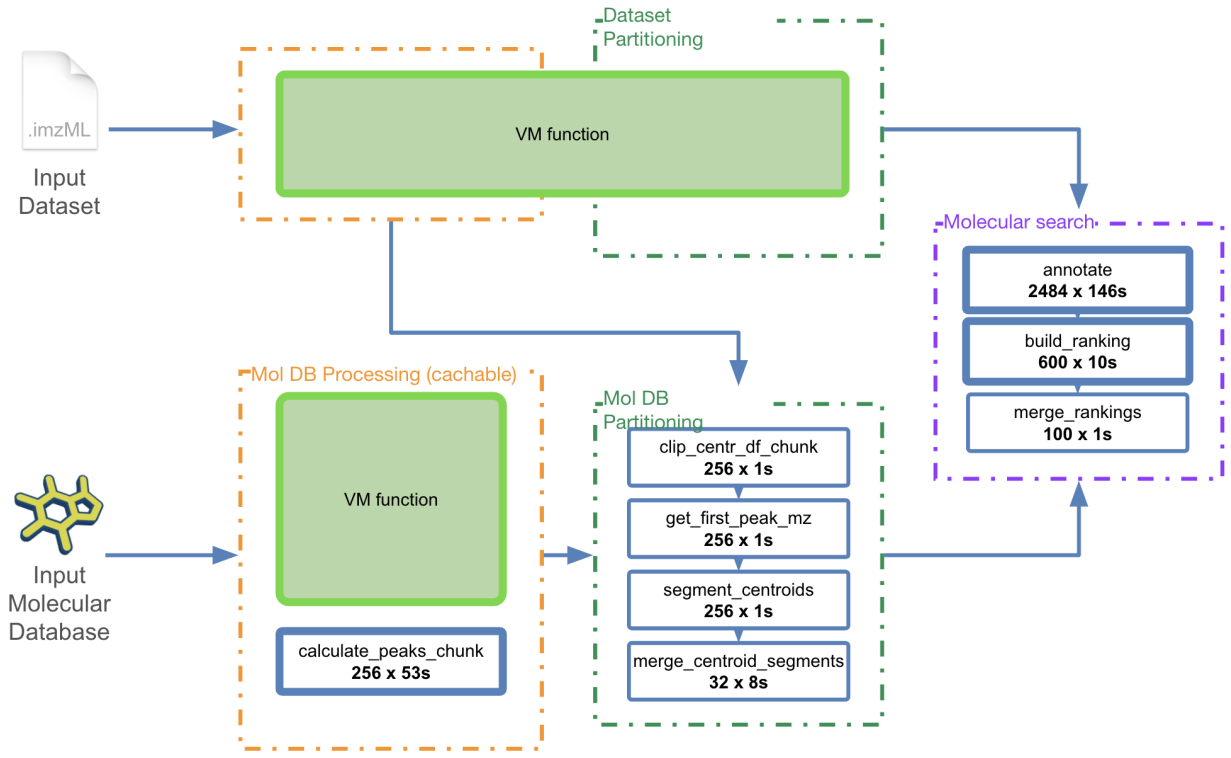


Figure 16: Hybrid implementation of the METASPACE involving using large VMs for the parts critical either for performance or for easy of development. The hybrid implementation is used since March 2021 in the production version of METASPACE.

Dataset	Size (MB)	Spark time	Lithops time	Relative time	Spark $USD	Lithops $USD	Relative cost
20190228_Rhodamine_Well3_p70s50_POS	63	709.633	331.423	-53%	0.342	0.094	-73%
NPC_179_pos	102	480.642	249.911	-48%	0.226	0.068	-70%
FDtest_exp7_mixed_slide_DAN_Slice2	592	496.158	265.631	-46%	0.234	0.079	-66%
DESI HEART SYNAPT-XS RES-MODE	1,245	630.096	576.766	-8%	0.301	0.266	-12%
150618-RatBrain-DHA-NEG-centroid	1,539	570.610	765.351	34%	0.271	0.251	-7%
2020-02-05_SlideD_DHB_POS_110x280_150umSS_31at	1,994	536.161	648.503	21%	0.254	0.258	2%
MPI//MPIMM_011_FT_P_KM	34,465	610.401	515.240	-16%	0.291	0.190	-35%
region1	39,733	744.605	839.450	13%	0.359	0.210	-42%
2019-12-19_DDN_microbe-spotting_exp2_45_400x900_30	41,532	3703.384	3544.381	-4%	1.853	5.777	212%
k233_combined three datasets	42,653	868.984	590.230	-32%	0.422	0.185	-56%

Figure 17: The results of benchmarking the Lithops implementation of METASPACE as compared to the previous serverful Apache Spark implementation.

the specified metrics.

The results of benchmarking the Lithops implementation of METASPACE as compared to the previous serverful Apache Spark implementation is shown in Figure 17. For the Spark implementation, we used the AWS hotspot instances for estimating the costs. One can see that the Lithops

implementation overall outperforms the Spark implementation in time and almost always but in one case for a very large dataset outperforms it in cost.

Moreover, the serverless technology and in particular Lithops provides a competitive alternative to Apache Spark in terms of the code readability and ease of development.

7 Genomics use case

The importance of genomics to our society could hardly be overstated. An increasingly large number of applications have genomics at their focus — among them molecular cancer treatments, personalised medicine, and combating viral diseases. The recent worldwide reaction to Covid-19, and the delivery of vaccines in a short time window, all hinged upon the availability of technology to sequence genomic material (nucleic acids) quickly and effectively (Figure 18).

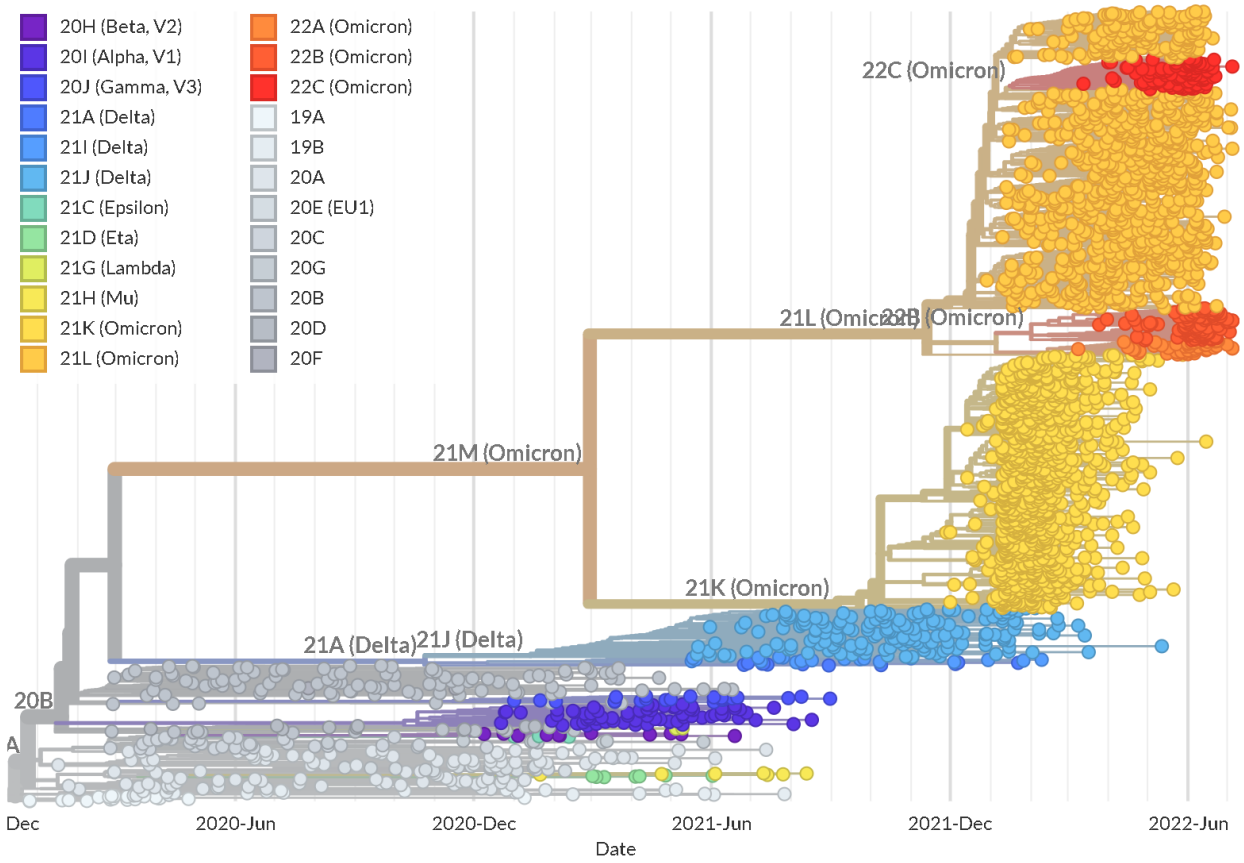


Figure 18: Phylogenetic tree of SARS-CoV-2, the virus causing Covid-19. Each point corresponds to a viral sequence obtained from high-throughput sequencing data (source: `https://nextstrain.org`).

Genomics is a data- and compute-hungry science, rooted as it is in sequence analysis. Genomes and their products (mainly DNA and RNA) are routinely sampled and decoded in order to establish how the cellular machinery works in different species. This has spurred the production of a staggering amount of sequence data, which results in several PB being added to global data stores every year (Figure 19).

In light of this situation, it is obviously very difficult for any single institution to be able to keep up. While high-performance computing (HPC) installations are becoming commonplace in biology departments, it is also increasingly challenging to provide enough capacity to support demand peaks, (re-)analyse public datasets produced by large consortia, or minimise the wall-clock time required by a given analysis when that is needed – for instance, when performing surveillance-related tasks in the domain of public health. The cloud would be an obvious solution to this problem, as it can supply enough elasticity to accommodate exceptional demand in HPC power without the need for scientific institutions to permanently acquire and maintain the corresponding physical infrastructure.

However, traditional cloud services are typically much more expensive than buying computing hardware, which discourages medium- and large-sized institutions from giving up their HPC clusters. With its offer of virtually unlimited computing power for a fraction of the cost of regular cloud services, serverless computing might provide a cheaper, more scalable and more attractive alterna-

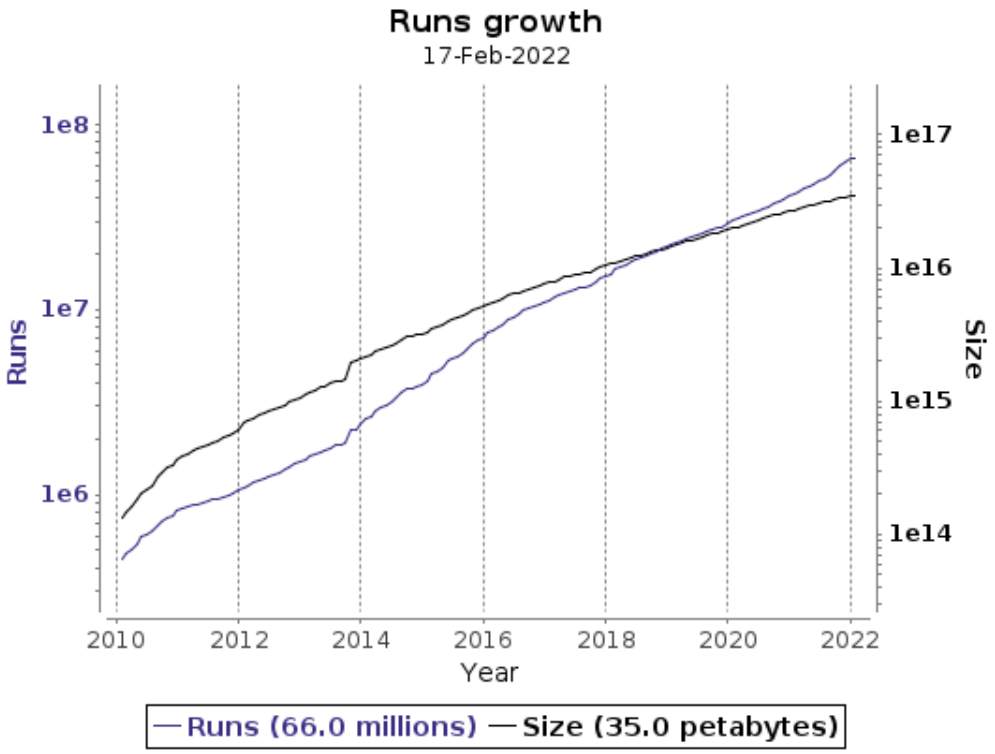


Figure 19: European Nucleotide Archive and Sequence Read Archive growth [71].

tive for genomics workflows.

However, transitioning to the cloud has been so far a complicated process. While genomics-centered solutions have been developed (for instance, [72]), they usually require the development of specific procedures to rewrite/encapsulate existing workflows. On top of that, evaluating and configuring the resources needed for each application is a daunting task due to the number of heterogeneous cloud providers and services.

Here we show how, by leveraging the power of the CloudButton toolkit, and Lithops in particular, we were able to demonstrate semi-transparent porting to (mostly serverless) cloud of a number of components and a workflow to perform high-throughput variant calling on genomic data.

7.1 Experiments description

The work we describe was performed as a collaboration of the James Hutton Institute, the partner in charge of the genomic use case, and other partners (Imperial College and URV). In the following, we will present four experiments:

1. **Porting of alignment tools to FAASM.** In collaboration with Imperial College, we explored how to port tools to perform alignment of sequencing reads to a reference genome, which is a frequent use case in genomics, to WebAssembly within the FAASM framework. This is a central problem, as it highlights the fact that genomic applications are usually stateful, and an adaptation to serverless frameworks requires adaptation of the existing code. We identified a solution consisting in the splitting of the large indexes needed for alignment into smaller ones.

2. **Development of general cloud toolkit components.** In collaboration with the Cloudlab at URV, we developed tools to tackle specific parallelisation tasks, i.e. FASTA and FASTQ file partitioning. Those are needed by most genomics workflows, and essential to port alignment tools to (serverless) cloud, as identified during the previous stage. They also include the implementation of a stateful solution to identify optimal sequence alignments across cloud functions,

and the implementation of a data-driven multi-function reduce architecture to integrate large alignments obtained from different functions.

3. **Porting of a variant calling pipeline to the CloudButton framework with Lithops.** In collaboration with the Cloudlab at URV, we used Lithops to port an existing variant calling pipeline based on [73] and [74] to a serverless-centric solution. This represents the main embodiment of our use case. It tackles a problem which is relevant in real-life cutting-edge applications (calling variants is an essential step in, for instance, personalised medicine, cancer treatment, or the typing of SARS-CoV-2/COVID-19 genomes) and demonstrates scalability, drastically reduced wall-clock time, and cost-effectiveness.

4. **Transparent conversion of legacy code.** In collaboration with the Cloudlab at URV, we developed tools to help with the transparent porting of existing code to the CloudButton framework, especially in connection with polyglot programming. More in detail, in this application we examine how a large codebase written in OCaml [75] and based on a local parallelisation model can be adapted to a (mostly) serverless architecture with a minimum number of changes, and provide the first ingredients needed to do so.

Our experiments aim to demonstrate a number of facts, which are captured in suitable KPIs:

- Prove that our existing genomics workflows can be successfully ported to CloudButton architectures, producing the same scientific results
- Demonstrate that CloudButton-based implementations can scale up to very large external datasets that we would not necessarily be able to store locally, or we would be unwilling to store locally long-term. That opens the way to obtaining new biological insights, for instance from the comparison between our local datasets and larger datasets publicly available
- Demonstrate that wall-clock time taken by CloudButton-aware workflows can be substantially reduced beyond what would be achievable on a large HPC node
- Evaluate efficiency/cost-effectiveness of cloud solutions versus local computing.

7.2 Integration of the genomics pipeline with FAASM

The first basic step performed by most genomics analysis workflows, and the first one we would like to migrate to the cloud, is called the *alignment* of sequence reads to a reference genome. By this we mean that, given a short chunk of DNA produced by a sequencer (a *read*), we want to find all the regions in the genome having a similar sequence, up to a pre-determined amount of differences between the read and the identified region. Equivalently, we want to locate all the regions in the genome that might have originated the sequencing read, as the accumulation of reads allows us to identify parts of the genome which are functionally active under different biological conditions (see Figure 20). Doing so is a computationally expensive operation, which usually takes most of the time used by analysis pipelines – the reference genome can be large and repetitive, one has to perform string searches with mismatches due to possible errors in the read or the reference, and billions of reads need to be processed, possibly more than once, for each experiment. The latter would suggest that alignment is a good candidate task to be ported to serverless.

Unfortunately, things are not so straightforward. First of all, alignment requires a highly optimised piece of HPC code. The code base for the GEM mapper [73] alone, which we would like to use for our workflows, comprises several tens of thousands of lines of C code, and other steps are implemented as a large library written in the OCaml [75] functional programming language. In addition, and quite unfortunately, some algorithmic requirements do not translate straightforwardly to serverless architectures. One example is the need for alignment programs based on the Ferragina-Manzini index, such as the GEM mapper, to generate and store into memory a binary data structure known as an *index*. The index is a transformed version of the reference genome; thanks to its design, it allows

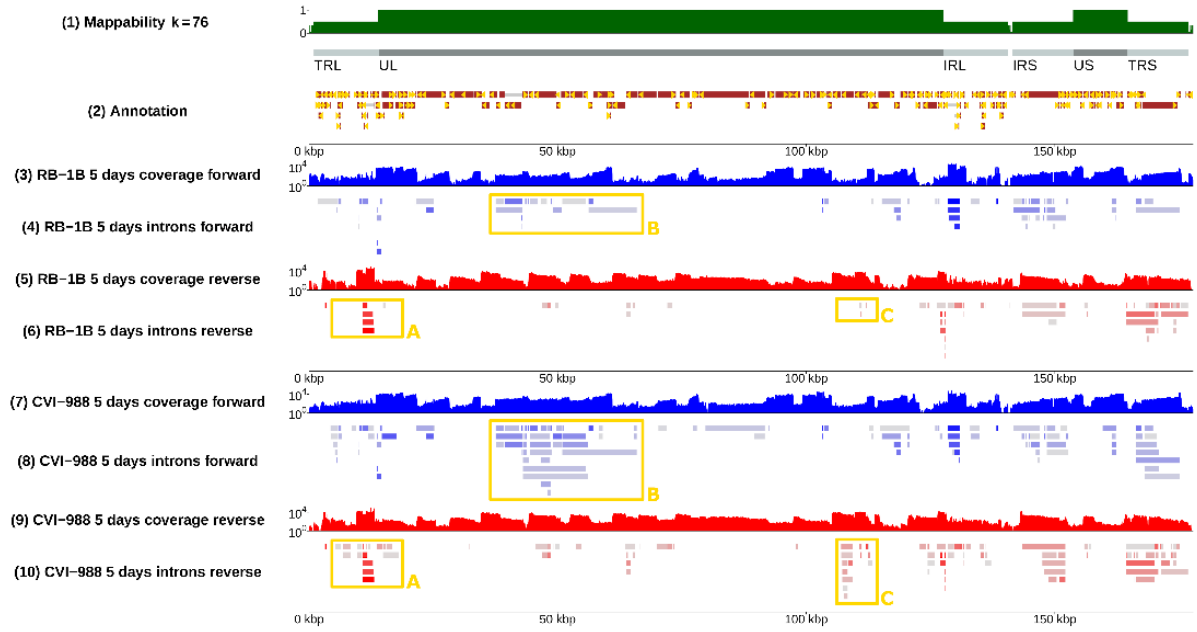


Figure 20: Transcriptional landscape of Marek's disease virus, a virus causing cancer in chicken and large economic losses to the poultry industry. The plot show the results of several RNA-sequencing experiments as a genome browser. Local accumulations of reads due to alignment produce a biologically relevant signal.

to quickly find exact queries in the reference, ultimately making possible the implementation of fast error-tolerant alignment algorithms for sequencing reads. Mammalian genomes such as the human one are relatively large (about 3 billion nucleotides) and as a result the index for a whole human genome can be bulky, ranging from several hundred MB to several GB, depending on the implementation. Unfortunately, such large amounts of memory are not generally available in a serverless framework.

FAASM is a high-performance serverless runtime that isolates functions using WebAssembly [76]. As a consequence, FAASM can only execute functions that can be cross-compiled to WebAssembly. Or, equivalently, languages with an LLVM front-end, and additionally limited support for Python functions [77]. While there is no support for OCaml in WebAssembly, we examined and discussed the best way to mitigate the problem of the statefulness of existing alignment tools for sequencing data, and came up with a solution suitable to be implemented in WebAssembly and FAASM. Such a solution is illustrated in Figure 21.

As is frequently the case for distributed computation, we can formulate our problem as two conceptual phases that fit the Map/Reduce model: *mapping*, and *merging*. During the first mapping phase, we split the sequencing reads into several *data chunks*, which are conceptually independent - as each sequencing read is aligned independently to the reference genome, this step is embarrassingly parallel. However, in order to reduce the amount of statefulness required by this step and mitigate memory requirements, we also split the genomic reference into smaller chunks. It should be noted that, while there is freedom in the amount of granularity adopted during this stage, splitting into *index chunks* is not trivially parallel, as the alignments of each data chunk to the index chunks will need to be collected and post-processed in the end in order to establish what the best ones are, and discard the ones having low quality – this merging step is performed automatically when all the reference genome is presented as a single index, but needs to be done externally if we want to split the big index into smaller chunks. Also, by splitting the index we are doing redundant computations at the expense of a higher degree of parallelism, which limits the amount of granularity we can reach.

As the code for the `gem3-mapper` is implemented in C, the mapping step can be ported to We-

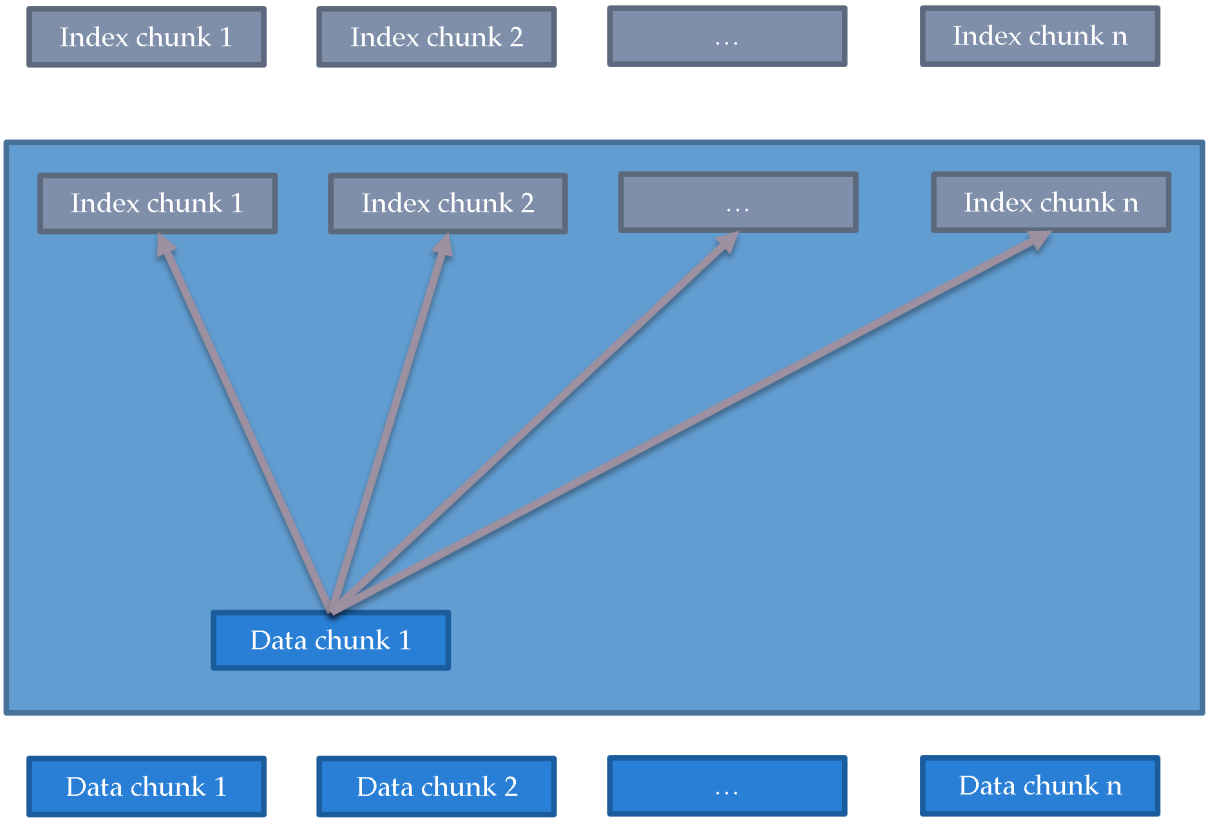


Figure 21: A scheme to parallelise sequence alignment and mitigate the problem of the stateful genomic index.

bAssembly using FAASM's compilation toolchain [78]. The forked repository that allows to cross-compile the GEM mapper is available on GitHub: `https://github.com/faasm/gem3-mapper`. Unfortunately, however, the code for the *merging* step is written in OCaml, and at the time of this writing we have not found a way to cross-compile OCaml to WebAssembly, thus hindering the adoption of this step in FAASM.

However, it would be possible to port the full mapping step with Lithops, offloading the mapping step to the serverless back-end provided by FAASM and invoking the merging step from Lithops. The whole pipeline integrated using CloudButton tools is summarised in Fig. 22. First, Lithops invokes the mapping function through its FAASM back-end ❶, then FAASM's mapping function fans out to several parallel calls to the `gem3-mapper` function ❷. Each mapper function reads its input pair (reads and index chunks) ❸ and write to their output file once they are done ❹. At this point, the merger code in OCaml, in a Python wrapper, is invoked from Lithops to merge all output files ❻, and write the single merged result into the object store ❼. Lastly, the merger can invoke the downstream pipeline ❽.

As a result of this work, we demonstrated that it is possible to port very complex genomic components, such as the alignment step of sequencing reads to a genomic reference, to WebAssembly with FAASM. We also identified a successful workaround (illustrated in Figure 21) to mitigate the problem of statefulness presented by the alignment stage. However, considerations about transparent portability of code suggested to us that a more general way forward to make genomic pipelines suitable for serverless computing would be by leveraging the flexibility of the Lithops framework. The next sections describe progress and results in that direction.

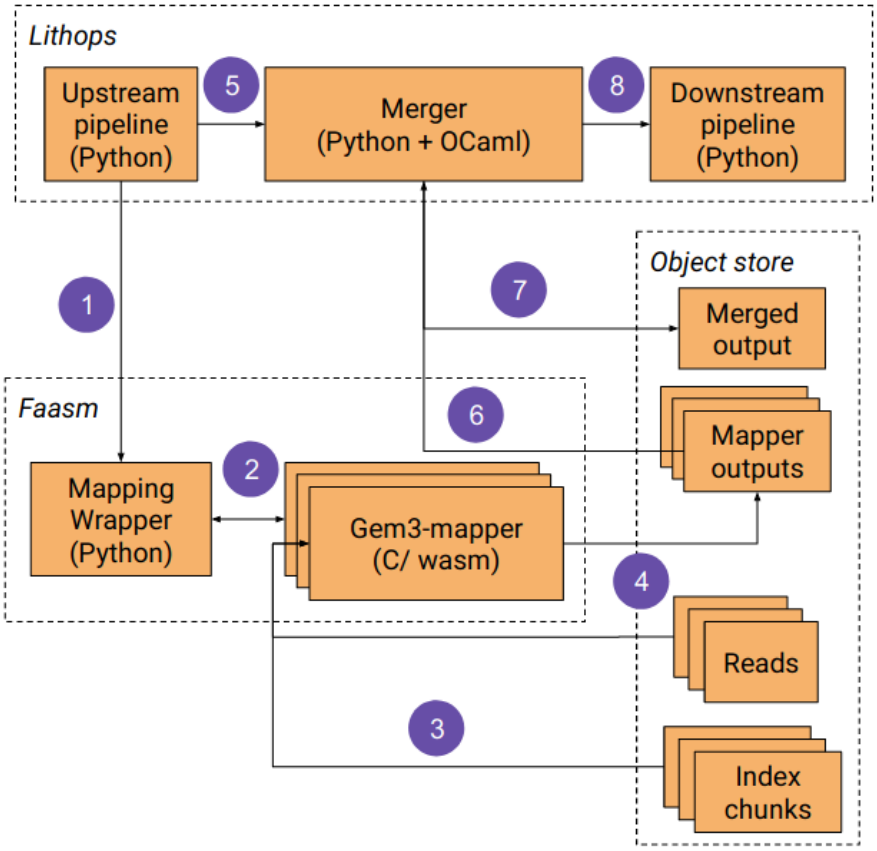


Figure 22: Integration of the genomics pipeline with FAASM and Lithops

7.3 General cloud genomic toolkit components with Lithops

7.3.1 FASTQ.GZ partitioner

Given its size, genomic data (such as FASTA and FASTQ files) is often stored in Gzip compressed files (.gz). The built-in data-processing logic of Lithops integrates a data partitioner system that allows to automatically split the dataset (usually a csv or a text file) into smaller chunks. However, the Lithops partitioner was lacking an implementation for compressed files, and thus did not support the partitioning of FASTQ.gz files, a key step in the parallelisation of the variant calling pipeline. Therefore, a Gzip-compressed file partitioner was developed (coded in python and bash), specifically for use within a Lithops `map_reduce()` implementation, though it can have other applications and also run locally.

gztool

A key component of the gzip partitioner is an existing software called gztool. Gztool is a GZIP files indexer, compressor, and data retriever, which creates small indexes for gzipped files and uses them for quick and random-positioned data extraction with no penalty. By default, Gzip-compressed files have not been designed to be accessed in a random way: to know the value of a byte at a given position x, it is necessary to decompress the file from the beginning to byte x. However the author of zlib, Mark Adler, developed 'zran.c', a cryptic file that creates an index of "windows" filled with 32 kiB of uncompressed data at different positions along the compressed file. The index can be used to initialize the zlib library and make it behave as if the compressed file begins there, thus allowing file chunks to be extracted and decompressed.

System design and processing steps

The partitioner operates in two steps, the latter involving either full or targeted partitioning and decompressing:

Step 1: Pre-processing. To decompress chunks of a compressed Gzip file, the entire compressed file must be decompressed beforehand, so as to generate the required index and associated data file for any given FASTQ file.

Step 2a: Full partitioning. This partitioning strategy is based on the division of the Gzip-compressed file into parts with an equal number of lines (except for the last one if the division is uneven), leading us to obtain all the chunks that make up the original un-zipped file. FASTQ chunks stored in S3 can be accessed by serverless functions directly.

Step 2b: Targeted partitioning. This partitioning strategy is based on decompressing a specific chunk, and can be implemented from within a map function to process a specific file range. This approach removes the need for storing FASTQ chunks in S3, as the original files can be processed on the fly. Thus, it also avoids the extra costs of communication to and download of files from S3.

7.3.2 FASTQ partitioner using SRAtools

The Sequence Read Archive (SRA) at NCBI is the main public repository of sequencing data. The latter is stored in AWS S3 buckets in a proprietory custom format and then converted to FASTQ during the download process. NCBI provides a toolkit called sra-tools that allows to download partitioned data from the cloud. The specific tool, fastq-dump, was deployed in parallel using AWS Lambda functions, to generate partitioned data accessible via S3 Buckets for downstream steps. The full workflow is as follows:

1. User provides an SRA sequence identifier
2. Sequence metadata is accessed from SRA, to establish the number of reads it contains without having to download the file.
3. The srasplit.sh script generates a list of chunks (byte ranges) of user-defined size
4. The list of byte ranges, included in the map function iterdata, is packaged for distribution to worker functions, orchestrated by the Lithops map function.
5. Each worker executes fastq-dump on an iterdata list item to download reads for a given byte range, and stores the FASTQ chunk in the worker's tmp folder.

This workflow was designed as described for three main reasons:

1. The fastq-dump implementation chosen accesses data via S3 Buckets containing sequences, not an external database, which makes the data transfer very fast.
2. fastq-dump allows part-file downloads, passing a range as shown, thus reducing per-worker bandwidth
3. Moving data between S3 and the worker micro-vm (or Lambda micro-vm) /tmp directory incurs no additional cost

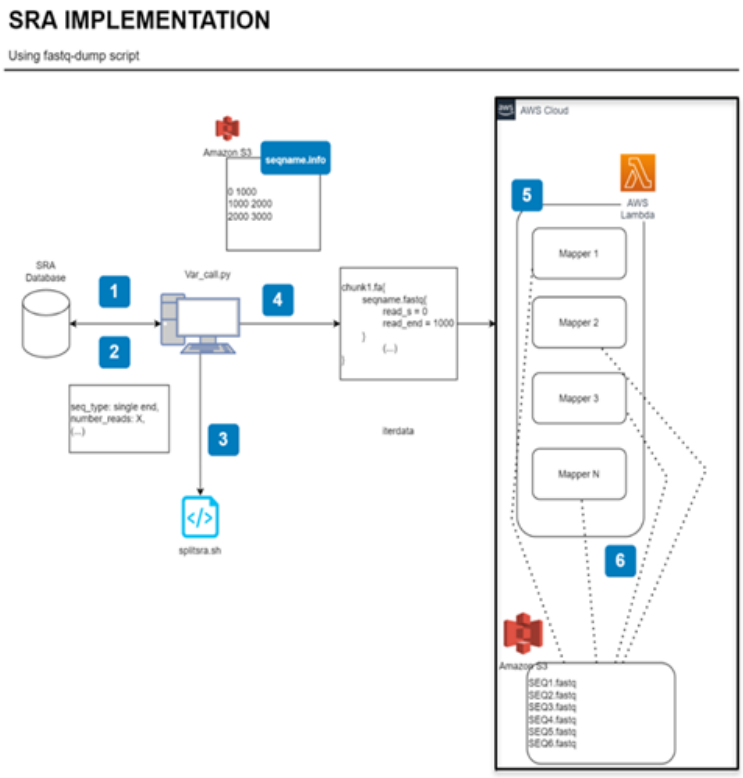


Figure 23: Schematics of the SRA partitioner.

7.3.3 FASTA partitioner

A distributed indexing and partitioning component for FASTA files was designed, to split large genome reference files (FASTA) into a user-defined number of chunks, each to be processed in parallel as part of the variant calling pipeline.

Splitting a FASTA file is not trivial, as the file contains multiple sequence headers (starting with >) and these are not evenly spaced, as chromosomes differ in length. Depending on the desidered chunk length, any given chromosome might be split into one or more subsequences, which in turn requires the addition of headers for each subsequence. For instance, `chr1`, split in two, would generate two headers, `>1_1` and `>1_1`. As per the example given, names are also simplified by enumerating them and the full chromosome name equivalent is stored in a table, for parsing at the end of the pipeline. This helps reduce the size of intermediate .mpileup files.

The partitioner workflow includes the following steps:

1. Generate a FASTA file index, containing the byte ranges of every header and every sequence (parall.)
2. Generate simplified headers for each sequence
3. Generate byte-range chunks, and split headers as required
4. Retrieve the appropriate byte ranges for each FASTA chunk, add the correct headers and save all FASTA chunks to S3.

Interestingly, the output of the first indexing stage is the same as the indexing produced by the `samtools faidx` tool, and therefore it was possible to compare the performance of these two indexing methods. The former adopts a parallelised workflow using serverless functions, whereby the index is the merged output of the parallel indexing of FASTA file chunks.

Name	Size	Distributed version	`samtools faidx`
Hg19	3 GB	13.00 s	26.9 s
Tb927_01_v5.1	25.7 MB	9.4 s	0.29 s

For smaller files, indexing is done better locally since there is no overhead caused by Lithops (mostly worker invocation and cold start), but for bigger files the advantages are obvious, with a 2.0x speed up. This advantage could be increased by using a smaller `obj_chunk_size`.

7.4 Integration of Variant calling pipeline with Lithops

Variant calling is a key step in most genomics pipelines, and most commonly it involves alignment of sequencing reads to a reference genome. Alignment mismatches are filtered and form the basis for establishing any mutations (variants) in the sample(s) analysed. Reference genomes are stored as FASTA files, each file containing several entries corresponding to a chromosome / sequence assembly. Sequencing reads are stored as FASTQ files, and contain both sequence data (like FASTA files), but also per nucleotide sequence quality information (hence the name FASTQ). In the genomics use case example we follow an established variant calling protocol, combining the gem3-mapper [79] for sequence alignment and SiNPle [74] for variant (SNP) calling. The purpose of this example is to demonstrate portability and scalability of this core genomics pipeline using the serverless architecture provided by Lithops. Lithops provides an extensible backend architecture (compute and storage) that is designed to work with different Cloud providers (such as IBM Cloud, AWS, Azure) and on-premise backends, allowing to run unmodified python code. Implementation of the genomics use case is centred upon Lithops `Map()` and `call_async()` functions, which are stateless Serverless Functions ran in micro Virtual Machines using AWS Lambda.

7.4.1 Pipeline overview

The variant caller pipeline uses Lithops as the orchestrator (using AWS as cloud provider) to implement a map-reduce framework, whereby FASTA and FASTQ files are split and all FASTA x FASTQ chunk combinations are aligned using the gem-mapper. Before alignment takes place, the FASTA file chunk needs to be indexed with the gem-indexer, which generates an indexed .gem genome file. The indexed FASTA chunk is used as the reference for alignment of a given FASTQ chunk, generating an alignment file, in .map format, which is then converted to the .mpileup format. While in the .map format each line stores alignment results for each read, the .mpileup format collects information from every read to provide a snapshot of the cumulative alignment results for each chromosome position, providing the number of reads supporting each base call detected across all overlapping alignments. The `mpileup` file is then converted to csv, and the file is stored in AWS S3. The reduce phase takes all .csv outputs from the map phase, calculates their total size and then proceeds to allocating the appropriate number of functions to process the data. Data processing involves merging `mpileup` data across sets of functions that share the same FASTA chunk as reference, i.e. those files that share the same chromosome positions. The merged mpileups are used as input for SiNPle, a Bayesian variant caller that calculates the posterior probability that every base called is true. The partial .sinple output from each reduce function is then uploaded to S3 and concatenated to produce the final output file. This can be used for any subsequent nucleotide mutation analysis workflow.

Figure 24 provides a schematic representation of the pipeline, followed by a more detailed description of each step in section 7.4.2. Porting of this pipeline to the cloud poses several challenges, which include the retrieval and accurate partitioning of FASTA and FASTQ files (as discussed in section 7.3, managing the size of the .gem index file, handling suboptimal alignments arising from mapping of any given read to multiple genome chunks (see also dedicated section 7.4.3, and the orchestration of multiple parallel reducer (discussed in more detail in section 7.4.4, to cope with the large datasets being processed.

7.4.2 Variant Calling pipeline: key steps

Below is a list of the key steps in the variant calling pipeline, which can be grouped into three stages, namely pre-processing (steps 1 and 2), map phase (steps 3-6) and reduce phase (steps 7 and 8).

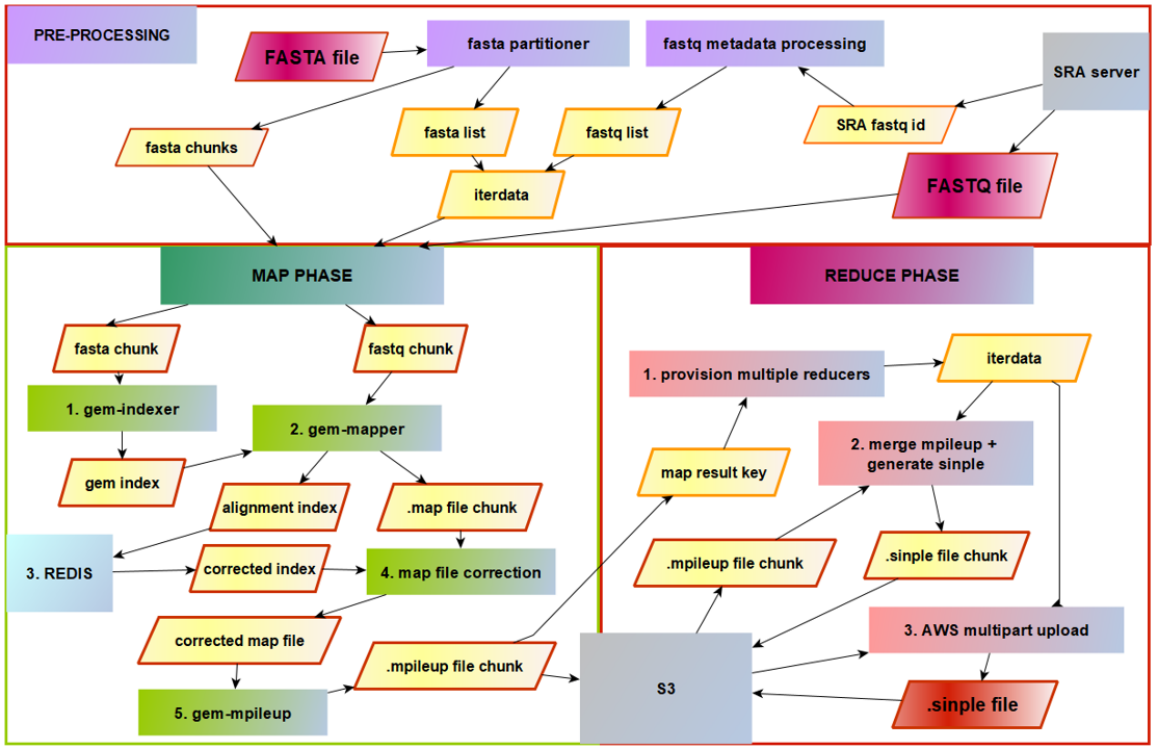


Figure 24: Variant Caller pipeline diagram

1. **FASTQ file partitioning.** Two versions of the FASTQ partitioner were designed, one that allows partitioning of "in-house" FASTQ files, gz-compressed and stored in a private S3 bucket, as discussed in section 7.3.1. The second allows partitioning of FASTQ files stored in the SRA database, and these are retrieved by the SRA toolkit, as discussed in section 7.3.2. Both tools calculate the correct byte ranges that allow partitioning of the input file based on the desired level of parallelisation (each byte range is sent to a separate map function that then retrieves the associated data). The smallest unit in a FASTQ file is a block of four lines (read name, sequence, spacer line and sequencing quality information), so, provided the frame is correct in the input file, partitioning the file is relatively straightforward.

2. **FASTA file partitioning.** The FASTA file partitioning process is more involved, as it requires replacement of original chromosome names with short identifiers, and, if necessary, the splitting of the same chromosome across different FASTA chunks, which in turn requires the generation of additional headers with simplified naming (i.e. >chr1 becomes >1, but, if split into two, would generate two headers, 1_1 and 1_2). Furthermore, an overlap needs to be kept between split chromosomes, in order for reads matching over the split not to be lost. The initial implementation required the FASTA reference to be downloaded and split locally, but an improved cloud-native version was designed that generates a FASTA file index, passing the index to lambda functions for parallel retrieval and storage of the desired-size FASTA chunks. This method was described more in detail in section 7.3.3. FASTA chunks are stored in S3 and then retrieved by the relevant map functions during the map-reduce phase.

3. **FASTA file indexing.** FASTA file indexing using the gem-indexer is required by the gem-mapper in order to optimise alignement performance, but it poses constraints on the size of FASTA file that can be processed by any given function: a 3Gb FASTA file currently requires a 13Gb index, which precludes the possibililty of eliminating the FASTA partitioning stage and partitioning only FASTQ files, given that the maximum storage capacity for lambda functions in AWS is currently 10Gb, and the maximum memory available to a function is 10Gb. With that said,

pre-indexing the FASTA file and allowing each map function to retrieve the gem file directly from S3 saves computation time and also allows larger genome reference chunks to be processed. Removing the indexing stage from the map function and importing the gme file has been a recent development (discussed in section 7.4.5, and it is hoped that it will help lower the number of FASTA chunks genereated in the first instance (down to 4-6 chunks from 18-20).

4. **Alignment of FASTQ file to FASTA file.** Sequence alignment is the key step that is parallelised in this variant caller cloud implementation, allowing wall-clock time to be brought down significantly (see 7.4.5). This step was optimised by abridging the gem3-mapper output, achieved by filtering all data reported back on reads with no matches and by adding some filtering to remove unlikely alignments. This additional filtration step significantly reduced the size of the intermediate .map file, and allowed simultaneous creation of an "alignment index", reporting the "score" of the best alignment for each read that passed the filter.

5. **Alignment index correction.** Whereas the pipeline as a whole embraces a serverless paradigm by employing functions to process sequencing data in parallel, splitting the FASTA reference into separate chunks (a necessity with the current workflow constraints, as indicated above) requires communication between functions in order to establish, for each read, the best alignment(s) across all FASTA chunks. Unlike the more widely adopted .sam alignment format, the .map format allows parsing of alignment "scores" for every reported alignment, thus making these easy to compare across FASTA chunks. In order for this comparison to take place, a stateful solution was put in place, namely a Redis database (`https://redis.io`) hosted on an EC2 instance, tasked with receiving indices from all functions, processing them for each FASTQ set (a set of functions sharing the same FASTQ chunk, and with different FASTA chunks), and returning the relevant corrected index to each function (with the highest scores for each read). The Redis index correction implementation is discussed in section 7.4.3. Index correction is complete when the previously filtered map file is filtered again to remove suboptimal alignments, i.e. any alignments with a score that is lower than the highest score obtained across all alignments associated with a given read.

6. **Generation of `mpileup` files.** Samtools provides a tool (mpileup) to convert .sam alignment files to .mpileup files, but such tool did not exist for .map files, so it was written specifically for the pipeline (and named gem-mpileup), once it became apparent that .sam files could not be used for index correction. `mpileup` metadata is returned by each lambda function and used to orchestrate the reduce stage, because the actual set of `mpileup` files is too big to be handled directly by any reducer; every partial `mpileup` file (mpileup chunk) generated is saved to S3 after conversion to csv.

7. **Merging of partial `mpileup` files.** The standard Lithops map-reduce framework is based on a single reducer function processing the output of every map function, which in this case would involve merging the base calling results for each genome position across all mpileups. This approach did not scale to the data sizes that needed to be processed by the use case. Next, a reducer was allocated to every FASTA set, thus processing every partial `mpileup` file belonging to a given FASTA set (i.e. all `mpileup` files that share the same partial reference sequence). This approach worked on small datasets, but, in consideration of the potentially very high number (hundreds) of FASTQ file chunks belonging to a FASTA set, reducers were allocated to FASTA subsets based on chromosome position ranges. In order to determine the appropriate ranges, file sizes were taken into consideration and then the appropriate split was determined, to avoid overloading functions with more data than they could process. Conversion from `mpileup` to csv (and then back to mpileup) allowed the data to be scanned using the AWS S3 select tool, to establish the appropriate ranges. This new multiple reduce implementation is discussed further in section 7.4.4.

8. Variant calling. The merging of partial mpileups, as discussed above, is the prerequisite for SiNPle to call nucleotide variants comprehensively for each genome position. Given that the algorithm used treats each base position independently, it was possible to allow `mpileup` merging and variant calling to be coupled processes running in each separate reducer. The final variant calling results are then "collated" from the output of every single reducer, firstly to reconstruct each FASTA set, and then the overall single file. This orchestration was achieved by using the AWS multipart upload tool.

7.4.3 Cross-function communication with Redis

The different processes or execution threads within a Lithops' Map_reduce() framework are executed in Serverless Functions or Virtual Machines. These Functions are stateless and thus not designed to communicate with each other. However, the Lithops framework allows each function to access the same storage backend simultaneously. In addition, cloud providers do not charge or have reduced costs for communication within the same area or datacenter. Therefore, we can orchestrate a communication system between the processes of the Map_Reduce() framework based on messages passed through the storage backend. If what is of interest is the storage of large files at low cost, one should use object storage services such as AWS S3, while, on the other hand, if the aim is to rapidly transfer smaller files, one can use databases such as Redis or Infinispan.

For this variant caller pipeline, the implementation choice fell on use of a virtual machine hosting a Redis server, acting as a storage backend for different Lithops clients, and processing the data gathered in its database. Redis (Remote Dictionary Server) is an in-memory data structure store, used as a distributed, in-memory, key-value database, cache and message broker, with optional durability. Redis popularized the idea of a system that can be considered at the same time a store and a cache, using a design where data is always modified and read from the main computer memory, but also stored on disk in a format that is unsuitable for random access of data, but only to reconstruct the data back in memory once the system restarts.

A library of functions was developed to automate the actions and control of the virtual machine and Redis database from a python client. The virtual machine that stores the Redis database and processes the intermediate files can be the same one that runs the Lithops client, so that communication costs are reduced.

7.4.4 Parallel reducer

MapReduce is a programming model for processing large inputs in parallel taking advantage of distributed systems. It consists of two phases:

1. In the map phase multiple map functions are called to process some key/value pairs and generate a new set of intermediate key/value pairs. Each mapper is assigned a partition of the total input data.

2. In the reduce phase a reduce function is called to merge, aggregate or transform all the intermediate values sharing same key. Each reducer takes its corresponding key/values from each mapper's output

There is an implicit phase between the map and reduce phase. This phase is known as Shuffle Phase and consists of the transfer of the mappers' intermediate output to the reducers. This phase helps reducers to easily distinguish when a new reduce function should start, as reducers can be triggered as map outputs are ready.

Lithops integrates a vanilla MapReduce function within its API. The problem with this function is that is not suited to processing large volumes of data, as it calls M mappers but a single reducer. This results in very poor performance that makes the scaling of architectures difficult. We therefore used Lithop's serverless function invocation APIs to create custom adaptations of the MapReduce model. The first multiple reduce prototype was developed in the IBM Cloud, and although it allowed `mpileup` files to be allocated to different reducers, it still did not allow scaling, given that merging

full `mpileup` chunks with others within a FASTA set (i.e. a set of `mpileup` chunks sharing the same reference FASTA chunk) eventually would lead to the creation of files that were too big to be handled by the reducers. So the next implementation (in AWS) was based on decoupling the allocation of `mpileup` chunks to reducers at the end of the map cycle: the `mpileup` file was now saved to S3 while the reduce phase received keys with metadata to orchestrate the retrieval of `mpileup` files or parts thereof within each reducer

With regard to efficient provisioning of reducers, at this point we should mention a further complication: not all `mpileup` chunks within a set will necessarily have reads at all genome positions included in that set, as the distribution of mapping reads will change from an `mpileup` file to the next. This implies that selecting a fixed range of positions to take horizontal slices of each `mpileup` chunk in a set could result in some reducers working under capacity because of the low number of positions to be merged for that subset. A first solution identified involved creating a dictionary storing all subset positions accompanied by the number of incidences of that position across `mpileup` chunks in a set. While this allowed for very accurate data-driven partitioning, it did create a bottleneck and posed a further scalability issue, given that dictionary capacity would limit the amount of data that any reducer could process. The current implementation relies on a first reduce stage that queries the number of lines within each range (the latter established based on an assumed even split of each `mpileup` file), followed by a filter that captures any intervals with very low numbers of positions to merge them with the previous/subsequent range, in order not to generate "near-empty" reducers. The advantage of this implementation is that it is also compatible with the multipart upload service offered by AWS, where partial files can be concatenated based on a list of keys provided, but this concatenation process is successful only if the partial files to be merged are larger than 5Mb.

The selection of specific positions or ranges in a file stored in s3 is made easier by a specific tool developed by AWS, called S3 Select. S3 Select is a AWS service that enables applications to get data subsets from a structured object on S3 by using SQL queries. Queries are run in the server side, releasing the user from SQL implementations, and are billed based on the amount of data processed. This service has some limitations, for example, the format of the file in which the queries are performed (needs to be .csv or .parquet) and the type of queries, which only allow filter operations.

Thus, the reduce stage was separated into two, one to query each `mpileup` file (after conversion to csv or parquet format) using S3 Select to establish the ranges that needed to be applied to each fasta set, and then a second reduce phase that allocated range subsets to different reducers, which then marged the `mpileup` subsets and generated .sinple outputs. The latter were first merged across subsets using AWS multipart upload to generate .sinple files corresponding to each fasta set, and then a further round of multipart upload was initiated to merge all sets into the final output file. A simplified diagram illustrating the overall architecture of this multiple reducer is given below in Figure 25.

7.4.5 Results

Integration and testing of pipeline components and their relationship with overall performance is still in progress, both in terms of optimisation of code for efficient use of individual lambda function capacity, and in terms of scaling requirements. Scaling has constituted the biggest challenge, especially in terms of ensuring reliable intermediate file orchestration between functions through Redis when the number of functions is large (>3000), and in terms of optimising the multiple reducer design to cope with large map function outputs. FASTA and FASTQ partitioners, on the other hand, have proven to scale well.

Here we present some promising preliminary benchmarking results, testing pipeline performance for a 101 Gb FASTQ file with 339M sequencing reads (SRA accession number: ERR9856489). Two similar data partitioning strategies were tested (hereafter referred to as run A and run B), with the human genome FASTA reference (∼3 Gb) split in 18 chunks in both cases, and the FASTQ file divided respectively into 260 chunks (∼385 Mb) and 225 chunks (∼444Mb). Given that every FASTA chunk needs to be aligned with every FASTQ chunk, this means that run A called 4680 map functions, while run B called 4050 map functions. Run settings were similar, but lambda runtime memory in run B

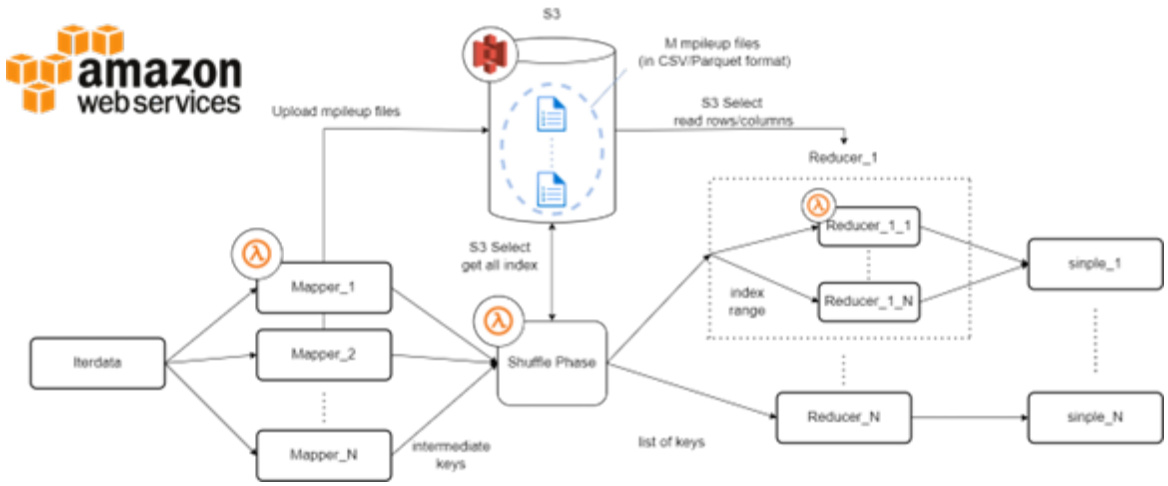


Figure 25: Multiple reducer architecture diagram

was increased from 4096 Mb to 5120 Mb.

Wallclock time

A first highlight of these results is the processing speed that can be achieved by using serverless functions, obtaining a high degree of parallelisation: the entire variant calling workflow was completed in ∼7 minutes (408 s) in the case of run A, and just over 3 minutes (188 s) in the case of run B. The additional memory allocated in run B dramatically lowered processing time, with a further gain constituted by the reduction in the number of functions. The wallclock time comparison (and associated AWS costs) between these two runs is illustrated in Figure 27 below.

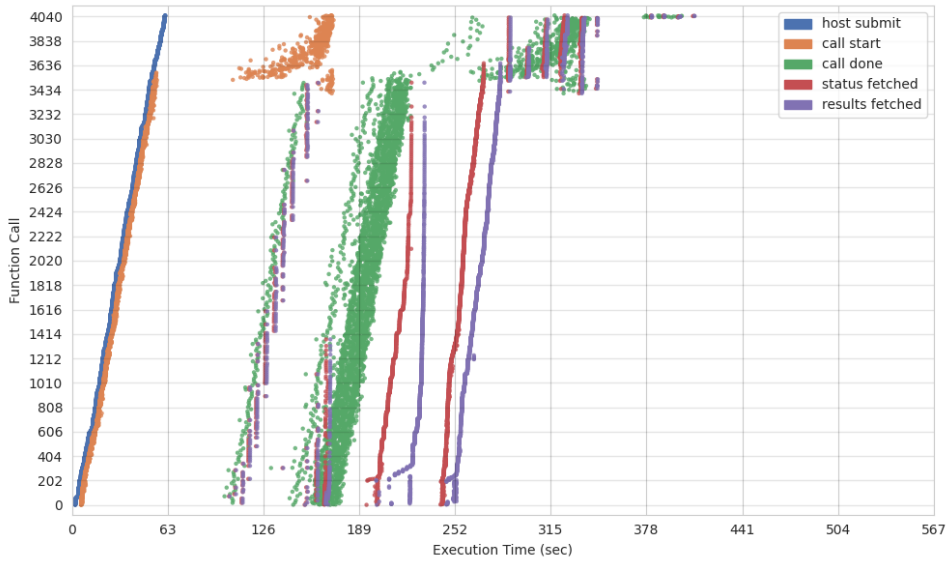


Figure 26: Timeline of a typical execution of this workflow.

Key performance indicators

As already discussed, it should be noted that the FASTQ dataset needs to be "replicated" as many times as the number of FASTA chunks, therefore, in terms of absolute data volume, in both runs we are processing 1.818 Tb of data. Measuring throughput for both run A and run B, we have achieved between 0.20 and 0.38 Gb/s throughput if we use the input file as measure, and 3.7 to 6.8 Gb/s in

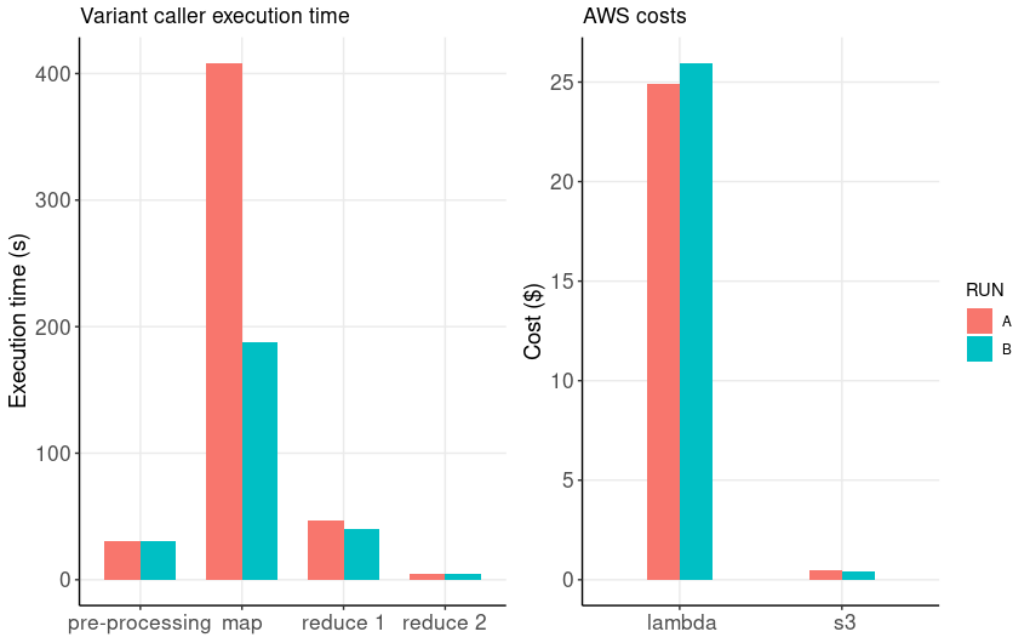


Figure 27: Wallclock time comparison of two variant caller runs (A and B) for FASTQ ERR9856489

terms of absolute data volume processed.

Scalability, elasticity

In terms of workflow scalability, in light of the absolute amount of data processed by the pipeline, due to parallelisation constraints, it must be stressed that additional improvements to minimise the number of FASTA partitions would ensure both less data processing redundancy and a lower number of functions used. This in turn would provide more scalability, i.e. the ability to process larger FASTQ files, by using a larger (but still feasible) number of concurrent functions, a cap on which is set by AWS at 10000 at the moment.

Cost effectiveness

For convenience, a lambda and S3 price calculator (PriceEstimator.py) has been included in the pipeline, reporting on the key costs associated with this workflow. It indicated that run A cost $25.5 whereas run B cost $26.5 (see Figure 27). These are comparable prices, suggesting in this case that faster execution can be achieved without any significant extra costs, as the gain in speed cuts down on execution time and therefore on Gb/s costs, and thus the faster run (run B) ends up costing the same as the slower execution (run A). The price of both runs was around 25 cents / Gb input file, which is quite high, but it can most likely be brought down with further pipeline optimisation. For example, a previous run with the same input files and partitioning strategy had double the cost, but this was successfully halved by removing the fasta indexing stage from the map function, given that fasta indices can be considered "fixed" and treated as inputs, rather than calculated on the fly for each run.

Productivity

While cost might pose barriers to adoption of this workflow until further optimisation and validation is completed, it should be noted that, in terms of wallclock time, this serverless variant caller outperforms the Dragen commercial services offered by Illumina, the leading sequencing company, which guarantees variant calling of a ∼120 Gb FASTQ file in 36 minutes (see and is considered the fastest variant caller available. By comparison, run A and run B completed processing 101 Gb in 3-7 minutes, which equates to 5-10 fold speedup. On a related note, one price estimate for using Dragen to process ∼120 Gb FASTQ was around $18 (https://www.lifebit.ai/blog/how-to-use-illuminas-dragen-

in-cloudos-ultra-rapid-highly-accurate-secondary-ngs-analysis-at-your-fingertips/), which equates to $0.15/Gb. Although our data is very preliminary and the pipeline still needs to be further developed, it could still be argued at this stage that the (less than) two-fold increase in cost for our pipeline compared to Dragen is offset fully by the more than 5-fold increase in speed.

7.4.6 Code, documentation and datasets

The code for the pipeline, and associated documentation, can be found at the following repository: `https://gitlab1.bioss.ac.uk/lmarcello/serverless_genomics/-/tree/main/variant_caller`. The main repository for sequencing data (FASTQ files) is the the Sequence Read Archive (SRA - `https://www.ncbi.nlm.nih.gov/sra/docs/`) and data can be accessed and downloaded by accession number. The data is stored in a proprietory compressed file format, and uncompressed to FASTQ format while downloading. Final FASTQ sizes are typically around 30-100Gb but can be as big as a few Tb. Various genome reference sources can be used, and are stored either at ncbi or in specific repositories associated with specific genome sequencing projects, depending on the organism. For our tests, we used the hg19 human genome reference sequence (∼3Gb), which can be downloaded from `http://hgdownload.cse.ucsc.edu/goldenpath/hg19/bigZips/`.

7.5 Transparent conversion of legacy code

The aim of this work is to study the transparent migration to the cloud of existing parallel components written in OCaml that have been extensively used to develop genomics data analysis pipelines. As a test use case, we will focus on the alignment collection stage mentioned in EXAMPLE 1 – the one that proved difficult to port to the cloud using WebAssembly and FAASM due to their lack of support for polyglot programming.

By *transparent* we mean that we would like to be able to compile and run in the (serverless) cloud our original code with as little modifications as possible, with all the complexity of distributed programming hidden from us, and with resource management and cost optimisation automatically performed for us. As a technology to fully do so does not currently exist, we focused on creating a programming environment that could be linked with the extant OCaml code with minimal modifications and provide sufficient performance.

7.5.1 Architectural breakdown and proposed changes

The original, single-machine multiprocessing model is illustrated in Figure 28. Having no need for network stack, it uses Unix fork to spawn processes and Unix pipes to communicate different processing stages.

The OCaml API for the framework is the following:

```
1 let process_stream_chunkwise ?(buffered_chunks_per_thread = 10)
2     (f : unit -> 'a) (g : 'a -> 'b) (h : 'b -> unit) threads
```

Listing 1: OCaml framework original main function declaration.

There are three main distinct prototype functions that must be implemented for each use-case:

1. The **chunking** function `f` reads the input and verifies its integrity before distributing the chunks to the worker functions. It is directly connected to all the workers.
2. The **map/worker** function `g` is directly connected to the chunking function to retrieve work and also directly connected to the reducer, in order to send the processed data. Multiple invocations of this function are done in parallel, using different processes and retrieving different workloads from the same chunker. Two workers will never process the same chunk even when running concurrently.
3. The **reduce** function `h` gathers workers processed chunks, preserving order and optionally saving state to generate final outputs.

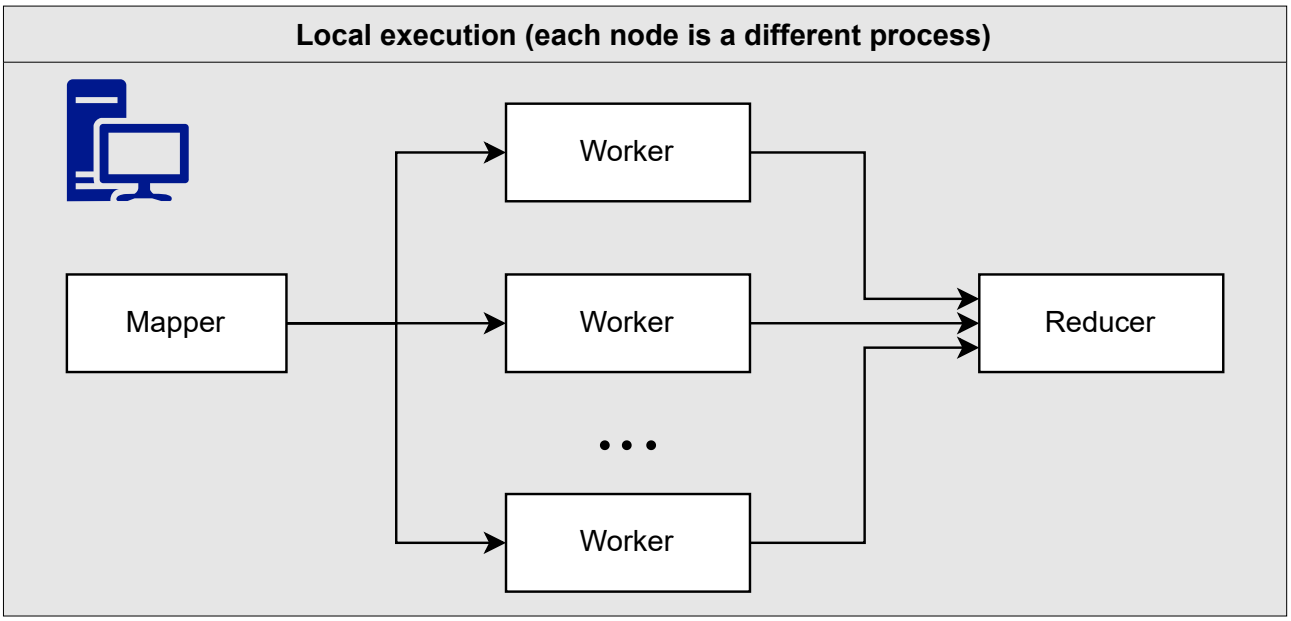


Figure 28: Original local execution architecture

The framework uses a custom stop-and-wait flow control based on OS pipes. Each process sends control bytes to start and stop reading the pipes at their ends, avoiding blocking the process when they are halted by the OS if the pipe memory buffer is full when multiple data blocks are sent. Also, the chunk size can be controlled by a parameter that is given to the chunker, so the pipe is never blocked by a single big chunk.

Main characteristics of the original framework are:

(1) There is no integrated shared memory mechanism.

(2) Forking processes allows leveraging virtual memory and Copy-On-Write kernel mechanisms to share variables from the parent process.

(3) Message passing arbitrary structured data between processes is possible thanks to serialization.

(4) There is always a single chunking instance and a single reducer instance, but an indefinite number of workers. The number of workers must be explicitly provided to the framework by the implementation code.

Point (3) has special significance for flexibility. OCaml being able to serialize most of its structured and primitive types, the sample program is able to send a tuple of a line counter and an arbitrary buffer type through pipes.

7.5.2 Adapting the OCaml framework with a Lithops Python wrapper

As Lithops is especially suited for highly parallel programs with little or no need for communication between processes, this use case is adequate due to only requiring communication between different phases of the execution (map and reduce, for which cloud storage can be used) and not direct communication among nodes.

Moving to the cloud should be as transparent as possible, with minimal code changes so that less code paths have to be tested again and the working logic is intact. In addition, function implementations have to be mostly treated as black-box logic because they should still work after the cloudification process. We will then assume that the basic building blocks of the frameworks (functions `f`, `g`, and `h`) are left intact and presented to Lithops as pre-compiled binaries. Lithops will then act as a high-level wrapper to provide localisation in the cloud and inter-process communication.

Inter-Process Communication

To provide communication between blocks, we use the `asyncio` Python framework. By leveraging it we are able to read and write to and from pipes using a single thread asynchronously from the python wrapper without complex protocols, without being bothered about thread overheads or the Global Interpreter Lock when reading pipes. Most work is I/O so CPU load is well distributed, reading and writing to pipes is a good use case for an event loop. In addition, other concurrent steps such as the multipart upload (see section **Reducer**) can be integrated into the same.

Chunker

We assume that the previous step in the pipeline saves the input file in cloud storage, be it a sequencer that uploads the sequenced data directly to the cloud or a preprocessing step.

The chunker may be adequate when processing local data because it is mostly I/O bound as it is always reading, up to the limits of the hard disk speed, there is no need to parallelize it. In contrast, it is inconvenient to chunk sequentially when processing data in the cloud. That is because the cloud is prepared for much higher aggregate bandwidth and parallelization levels, being able to read different sections of the files in cloud storage at the same time. It is also fundamental to partition based on cloud storage and not to use a local chunker outside the cloud network as that could create a bottleneck based on the client available bandwidth and packet loss while uploading the chunks to the workers.

In order to remove such a bottleneck, we base our revised, cloud-ready chunker on a byte-range partitioner, with the ability to read in parallel from the cloud storage different chunks. That is not entirely straightforward due to the presence of structured records in the input file. We correct that with a step based on regular expressions that locates the end of the first record overlapping the boundaries between any two blocks. Two example regular expressions are provided for two different formats: FASTQ [80] and MAP single end reads [73].

The algorithm works as follows:

1. We generate a set of evenly sized byte ranges so that the file is entirely split by them. The last byte range might have a shorter size than the others.
2. We append a predefined number of bytes at the end of each range (bytes that overlap with the start of the next range). That is not needed for the last range.
3. We invoke a function call for each byte range.

After the partitioner invokes the function call, the following happens:

1. The Lithops wrapper map implementation code reads the byte range entirely to memory and pipes it into the standard input of the OCaml binary.
2. The OCaml code seeks the first match of the provided regular expression for that format. When the first match is found that position is saved as the start of the real chunk, discarding anything before it.
3. The OCaml code seeks the first match of the same regular expression but this time at the end of the chunk, just before the extra window at the end. The matched position is the end of the real chunk, it may be inside the extra window at the end.
4. Now the chunk is already healed, entirely in memory and the local chunker may distribute the work among the worker processes.

To avoid having Lithops as a dependency when developing locally, a OCaml partitioning scheme that mimics the Lithops partitioner with local files was also developed (byte range generation) and added to the framework toolset. It does not require any Python interpreter to run.

Worker / Map

As the original framework does not make any hard assumption on the nature of the workers, one might be tempted to think that a good solution can be achieved by just replacing Unix pipes with cloud storage. Unfortunately, that is not the case. In fact, preserving the local multiprocessing code is important to reap the benefits of vCPU scaling, which solely depends on the memory runtime chosen for the cloud function call – the higher vCPU tiers provide three, four, five and up to six CPU cores, see Table 2. And, if running inside a virtual machine instead of cloud functions, local multiprocessing is also useful.

Memory	*vCPUs*	*CPU Ceiling*	*Memory*	*vCPUs*	*CPU Ceiling*
832 MB	2	0.50	5308 MB	4	2.67
1769 MB	2	1.00	7076 MB	4	2.84
3008 MB	2	1.67	7077 MB	5	3.86
3009 MB	3	1.70	8845 MB	5	4.23
5307 MB	3	2.39	8846 MB	6	4.48
			10240 MB	6	4.72

Table 2: Experimental analysis of AWS lambda vCPU scaling [81].

That is why we modified the local chunker in the OCaml binary to ingest the whole chunk at once and distribute the chunk to the local worker processes. This way, Unix pipes are still used in the local execution inside the function to scale on vCPU usage and avoid being billed for idle compute. If a particular use-case required a single process/worker, with the proposed architecture, the same code would still properly operate with no modifications needed.

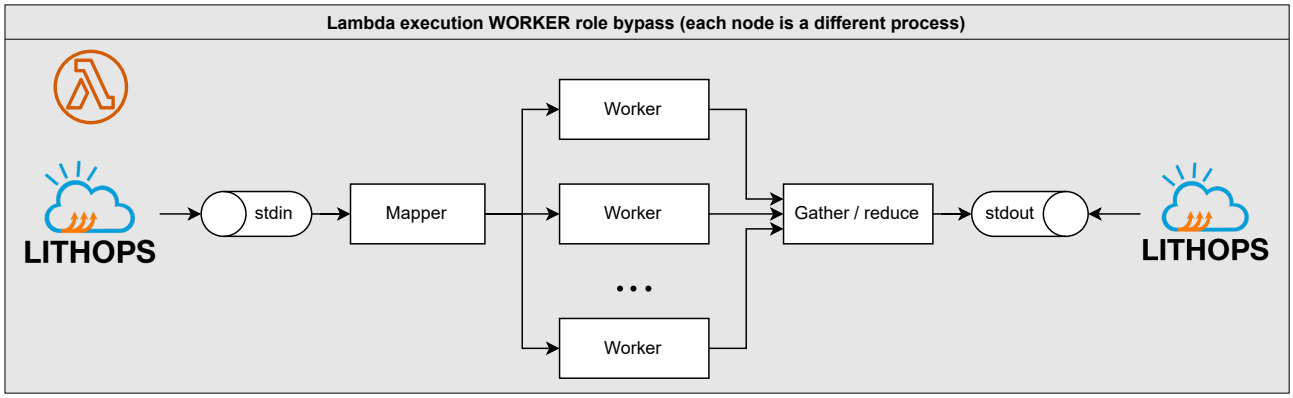


Figure 29: Worker using Lithops wrapper

On the new architecture, while executing the worker role, the local reducer collects the outputs of the local multiprocess workers as if it was to reduce them, but instead, it redirects the serialized data to standard output. Meanwhile, the Lithops wrapper consumes from this standard output pipe and uploads data in chunks to the cloud storage bucket.

Reducer

The reducer is the biggest bottleneck in the whole architecture. By design, it is not parallelized, launching only a single process for each program execution. This seriously impairs scalability on modern architectures, but is necessary due to genericity constraints – implementing a distributed reducer architecture might impair the capacity for the reducer to be used with generic legacy code.

In our specific use case, it is still possible to speed up the final reduction process by performing a binary reduction of the chunks.

A multipart upload allows aggregating a single object as a set of parts. Each part is a continuous portion of the object's data and can be uploaded independently or without preserving the original order. This feature has a lot of benefits, but we are only interested in a few of them, such as the ability to begin uploading an object without knowing the final size. What is particularly interesting to us in this case is the capability to stream parts sequetially. Saving each mapper's output to different files, aggregating these paths to a list, providing the reducer with the list and the reducer fetching each file individually in order, is the best option for simplicity. No coordinator is needed for this and functions still benefit from streaming (although to separate files) in the framework's architecture.

The library used for this task is called *aiobotocore* which exposes an async API for the *botocore* AWS library. In this case, no extra configuration is required other than installing the dependency using the *pip* package manager. A simple modification is done to the code so that Lithops hands the credentials used to the *aiobotocore* client and no extra authentication must be set up.

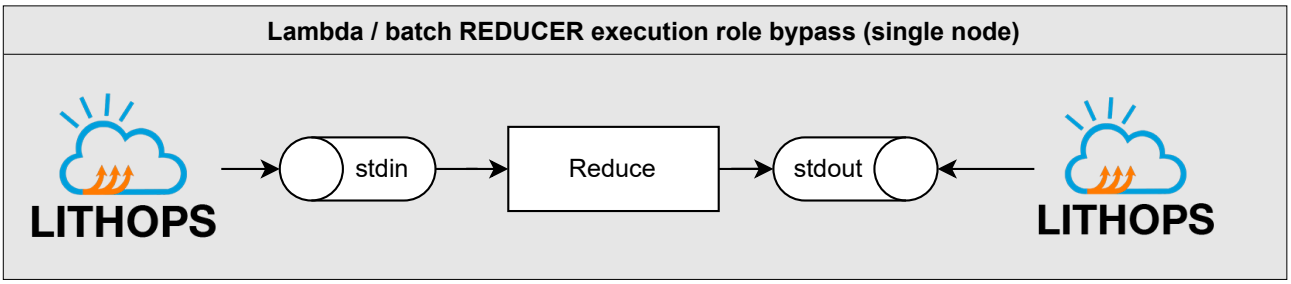


Figure 30: Reducer using Lithops wrapper

Now the Lithops wrapper gathers all the chunks saved by the workers from cloud storage, preserving order and piping them into the standard input of the OCaml binary one after the other.

The output from the reducer is uploaded to a single file. However, the OCaml reducer binary is still free to use any available cloud storage libraries to directly write an arbitrary number of additional files to cloud storage itself.

Program execution flow

Instead of the chunker directly reading from files using OCaml standard library, it should receive its input via standard input as a string and distribute it to its child processes.

The redirection of the local reducer input to standard output when running in a map function comes at a cost: a code refactoring/constraint. The wrapped binaries must decide when to take the reducer code path that redirects the input to standard output and when to reduce the input directly (do the task of the actual reducer implementation).

A cheap and quick way to differentiate when to execute a worker or reducer is by using environment variables, set by the Lithops map and reduce implementation functions to differentiate code paths to run depending on where the same binary should take the role as a reducer or a worker. The resulting switch can be transparently embedded into the modified OCaml API. Another advantage to this solution is that environment variables can be queried from any binary independently of the programming language used, providing language transparency in the mechanism that the wrapper uses to communicate which code branch is to be run (worker or reducer).

Ocaml API changes

The proposed strategy results in a modified API that allows OCaml code to interact with the Lithops wrapper environment:

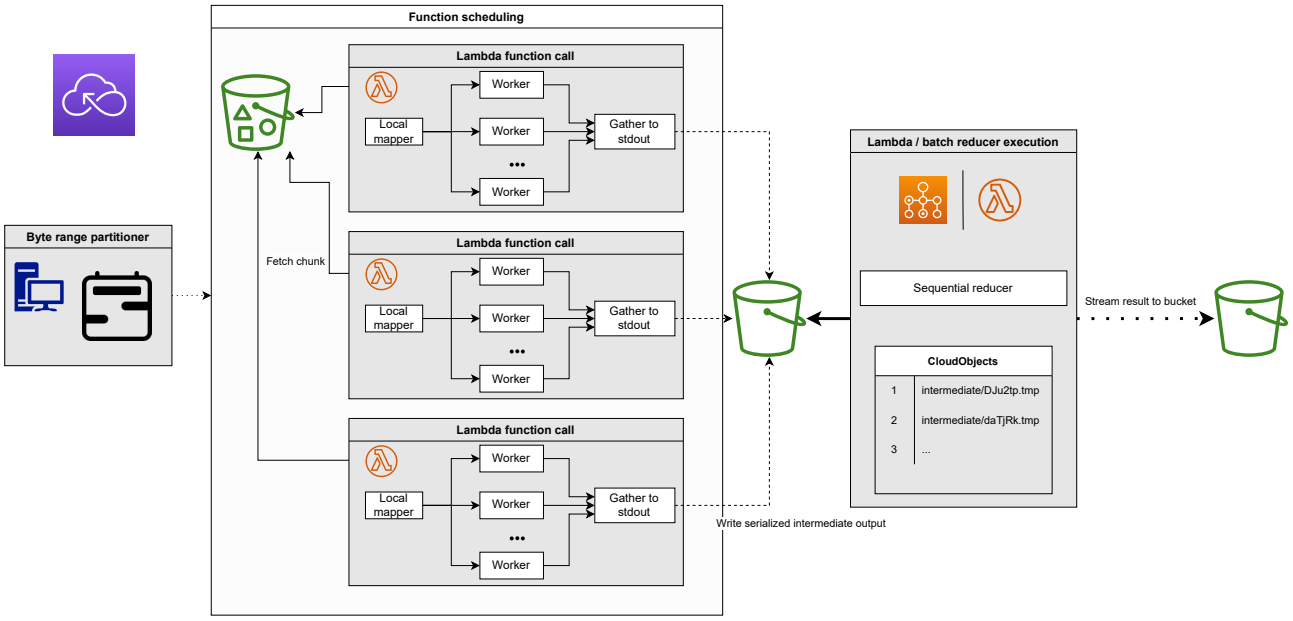


Figure 31: Cloud framework architecture

```
let process_stream_chunkwise_with_Lithops ?(chunk_size_in_mb = 32)
    (filename : string option) (elem_detector : Str.regexp)
    (chunker : string -> 'a) (worker : 'a -> 'b) (reducer : 'b -> unit)
```

Listing 2: OCaml framework adapted main function declaration.

7.5.3 Validation

The purpose of this section is to:

- Highlight streaming from memory to the cloud storage without configured temporary storage.
- Emphasize on the throughput gained from cloudifying the map phase.
- Demonstrate how the performance scales as the input file gets bigger.

The speedup is mostly expected on the Map phase as a result of the Map function completely running in parallel without communication between nodes (embarrassingly parallel), being the Reduce phase equal or worse than in a single-machine setup (sequential). Leveraging serverless functions for the map function, there is the benefit of aggregate bandwidth and scalability, whereas the reducer can just use a virtual machine or a cloud function if the file is small enough.

Tests have been prepared to generate throughput and speedup plots for the map phase, using variable volumes of 4 and 100 GB. The files are saved in S3 cloud storage and the Lithops coordinator runs from a laptop using wireless outside the cloud internal network.

As baseline for testing single-machine performance we processed files through a *c6id.2xlarge* instance. C6i instances are powered by 3rd Generation Intel Xeon Scalable Ice Lake processors. This instance in particular has 8 vCPUs, 16 GB of memory, up to 12.5 Gbps of network bandwidth and a 474 GB NVMe SSD. The input files have been downloaded from S3 to the hard disk to speed up the reading phase. However, the outputs are directly streamed to S3.

Figure 32 shows that the large file gets a limited speedup of $\sim 2\times$ on the VM, even when many processes are used (results are similar for the smaller test file). On the other hand, Figure 33 showcases how using serverless functions and their aggregate bandwidth can scale better for huge files thanks to a parallel architecture.

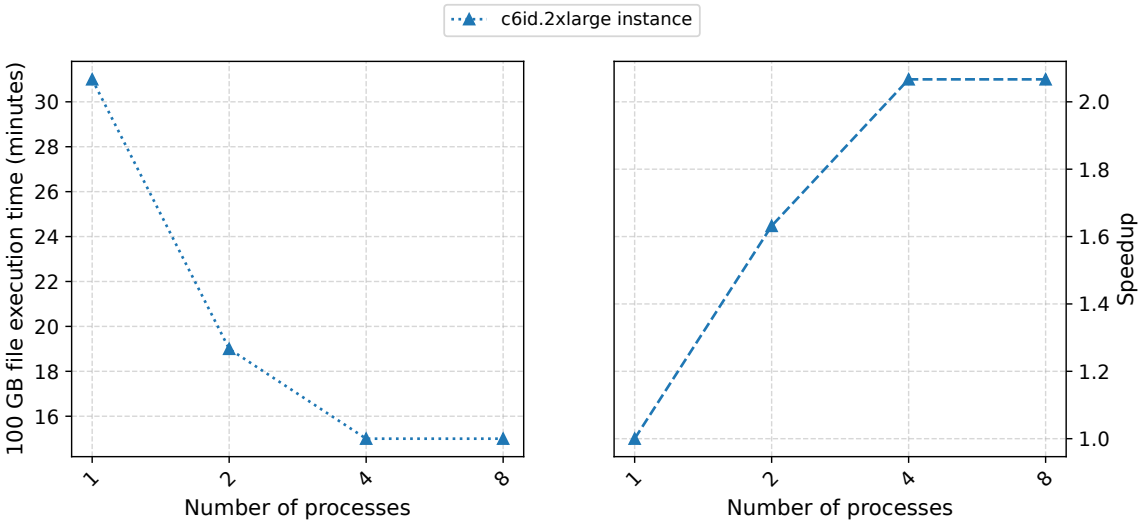


Figure 32: Virtual machine execution time and speedup, 100 GB file

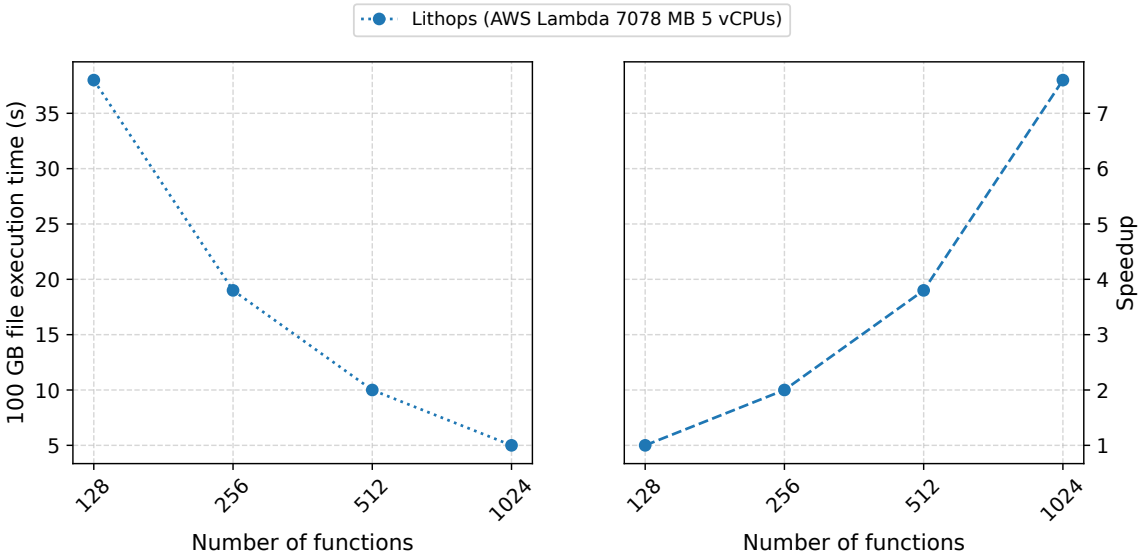


Figure 33: Lambda execution time and speedup, 100 GB file

It is important to consider that the tested Lambda configurations are not the fastest available and the concurrency quota might be increased to reduce wall-clock time even further on large files.

The total billed amount for the *Lambda* experiments (all configurations and plots) was approximately $5. Temporarily storing results in S3 cost less than 40 cents.

Key takeaways

Our work has led us to conclude that an inter-process communication wrapper is the simplest approach towards the cloudification of an existing multiprocessing parallel framework already used in different genomic use-cases and written in OCaml. By treating each program like a black box instead of tweaking its internals in a lengthy process, we have devised a satisfactory methodology to achieve transparency and avoid the overhead of rewriting business logic. After the changes, the new architecture outperforms single-machine executions for huge files and the local-cloud performance gap greatly improves the bigger the input files are.

7.6 Conclusions

The genomics use case has demonstrated portability of a variant caller pipeline to the cloud, and has spurred the development of tools and workflows at the interface between genomics and cloud computing. Genomic processes such as genome indexing and mpileup file generation were adapted to meet the requirements of a parallelised workflow, leading to the development of a serverless solution with a stateful component to manage optimal sequence alignment selection across functions. This, in parallel with dedicated data partitioning tools and carefully designed function orchestration at the map / reduce interface, allowed scaling of the variant calling pipeline to >100Gb input files, obtaining results in a very short wallclock time (3-7 minutes), which is way quicker than commercially available alternatives and has a comparable cost. This performance level is already sufficient to process commercial data routinely produced in the field; with further consolidation and refinement of map and reduce function throughput, it is foreseeable that even larger input files can be processed with similar run times, at a commercially competitive cost. This paves the way for elastic bioinformatic workflows that are controlled from a laptop and allow anyone, including labs with limited resources and not possessing specific expertise in the field, to assemble on-demand large amounts of CPUs and resources in the cloud for the timely and cheap processing of their data — or for the fast reanalysis of large datasets already available in the cloud. We also explored in some detail the question of portability of legacy code, possibly written in unusual or polyglot languages, and how bioinformatics workflows should be optimised to make them ready for cloud environments. In general, the cloud requires bespoke solutions and can help drive genomics to the adoption of file formats and algorithms allowing faster and more elastic querying and processing in a distributed environment. Successful development of reusable bioinformatics components, a variant calling pipeline and a parallelisation engine based on Lithops and presented as a transparent OCaml API suggest that other bioinformatic workflows could also be ported to the cloud, building on the tools developed in this use case.

8 Geospatial use case

Geospatial data [82] can be described as data that provides geolocated and temporally defined information about some aspect of the earth's surface and its characteristics.

Geospatial data [82] is very diverse and is obtained from dispersed sources, defined in a variety of data type formats. Examples of geospatial data could be polygons delimiting land surfaces, with added metadata such as census data by urbanization, or spatially dispersed points representing weather stations producing weather data continuously over time, or satellite imagery represented as arrays of thousands of cells describing the surface of the Earth.

In [82], they list the following kinds of geospatial data:

- **Vectors and attributes:** Descriptive information about a location such as points, lines and polygons.
- **Point clouds:** A collection of co-located charted points that can be recontextured as 3D models.
- **Raster and satellite imagery:** High-resolution images of the Earth, taken from above.
- **Census data:** Released census data tied to specific geographic areas, for the study of community trends.
- **Cell phone data:** Calls routed by satellite, based on GPS location coordinates.
- **Drawn images:** CAD images of buildings or other structures, delivering geographic information as well as architectural data.
- **Social media data:** Social media posts that data scientists can study to identify emerging trends.

Geospatial analytics [82] is the process of analyzing geospatial data in order to extract useful information to create data visualizations and to incorporate timing and location attributes to other conventional sorts of data. These visualizations can highlight historical changes and present shifts using maps, graphs, statistics, and cartograms. Visual patterns and pictures that are simple to recognize give insights that otherwise could be missed in a large spreadsheet. This might lead to quicker, simpler, and more accurate forecasts.

Geospatial data analytics present two main challenges:

1. First, **geospatial data occupies an extremely large volume of space**. For example, it is estimated that 100 TB of weather-related data is generated daily [82]. Data management and efficient storage and access is still a big issue for this magnitude of volume.
2. Second, **geospatial data is represented in many different data formats**. Geospatial data analytics requires complex pre-processing to prepare data in order to just present it efficiently and in the appropriate format to the requesting analytics application. These processes are, for example, data format transformation, filtering, partitioning, alignment...

These challenges have been studied in the context of Cloudbutton project with Serverless technologies in mind. Serverless technologies can facilitate the implementation of geospatial analysis workflows in the Cloud and overcome technical limitations, like scaling, that other solutions (like on-premise processing) cannot handle. In brief, we take benefit from the high scalability and flexibility of serverless functions combined with the high bandwidth capacity of Object Storage parallel read in order to reduce execution time and increase speedup by parallelizing the geospatial analysis processes. Geospatial processes are trivially parallelizable: the more land surface to analyze, the more independent parallel processes we can run in serverless functions. Nonetheless, one problem we have faced in this use case is the preparation of the data. In order to provide the functions with optimally partitioned data to effectively perform the parallel computation, the data must first be preprocessed, transformed, partitioned and sanitized. Other data formats, like CSV tabular data, are

trivial to partition (as data can be split by number of rows). However, geospatial data formats are not prepared to be directly consumed partitioned in parallel from Object Storage: most of the data formats expect a file system interface where the application can seek and read data chunks with low penalty. Thus, geospatial data formats are a particular case in which pre-processing and partitioning are decisive when looking for partitions of optimal size and obtaining a good performance in the data analysis phase. In this use case we will place special emphasis on this preliminary phase of the analysis, since the subsequent tasks are of no interest with regard to novel challenges of parallel analysis using serverless functions.

8.1 Geospatial use case: a general overview

The geospatial use case consists of different individual workflows. However, many workflows have data dependencies between them. Figure 34 represents the big picture of the whole use case, where we have integrated the different geospatial use case workflows. We can see the different inputs and outputs each workflow has, and the data dependencies between them. In the next sections, we will describe each workflow separately, providing insight on how data management is handled utilizing Cloud Object Storage and serverless functions.

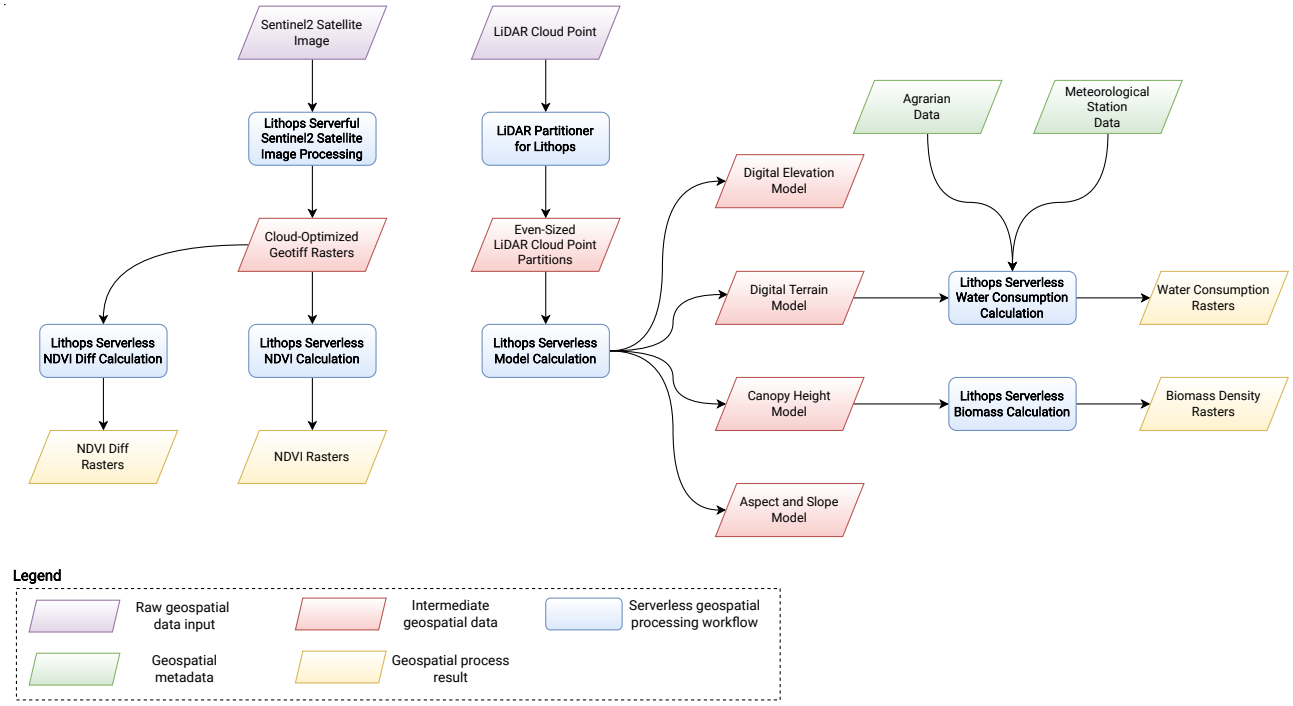


Figure 34: Geospatial use case overview.

The purple rhomboidal boxes represent raw input data, obtained from open data portals such as the Spanish National Center for Geographic Information (Centro Nacional de Información Geográfica, CNIG) or Copernicus Open Access Hub. The green rhomboidal boxes identify geospatial metadata, generally point data stored in tabular formats such as CSV. Finally, the red rhomboid boxes represent intermediate data between workflows, and the yellow rhomboid shapes represent the workflow outputs – results that can be analyzed by experts in order to draw conclusions about the problem at hand.

Blue boxes represent geospatial workflows utilizing serverless technologies such as Lithops. Here we would like to remark that Lithops provides the ability to both **preprocess** data for transforming and filtering and to **process** data to obtain conclusions. As mentioned before, it is crucial to preprocess the data in order to provide parallel reading capability from many functions, so as to exploit the parallelism of serverless functions. In this case, the **preprocessing** workflows are *"Lithops Serverful Sentinel2 Satellite Image Processing"* (Section 8.4), *"LiDAR Partitioner for Lithops"* (Section 8.2) and

"Lithops Serverless Model Calculation" (Section 8.3). On the other hand, the data processing workflows are *"Lithops Serverless NDVI Calculation"* (Section 8.5), *"Lithops Serverless Water Consumption Calculation"* (Section 8.6) and *"Lithops Serverless Biomass Calculation"* (Section 8.7).

For each workflow, we will briefly review what the geo-processes consists on, how data is managed and partitioned, how it is scaled and parallelized using Lithops and serverless functions and finally which is the impact of each workflow implementation in terms of key performance indicators.

8.2 Preprocessing Workflow: LiDAR Partitioner for Lithops

This section explains the development of a tool to partition LiDAR files. LIDAR data is the base from which, through preprocessing methods, we can extract terrain models that are used in other workflows (see Section 8.3). Therefore, the main motivation for partitioning LIDAR files is to increase parallelism and speedup by using serverless functions and thus reducing the overall workflow run time. We start by introducing the main characteristics of LiDAR files and their particularities. Next, we explain the problem of partitioning this type of files. Finally, we detail how the tool has been implemented and evaluate its efficiency.

8.2.1 LiDAR file format

LiDAR files represent a cloud of points, generally in a 3D space with (x, y, z) coordinates among other metadata, for example RGB color of the point. Point clouds obtained with LiDAR (Light Detection and Ranging) systems can be stored using different types of files. Among the most common we can find: generic ASCII files, LAS files [83] or LAZ files (also named LASzip) [84]. In this work we focus on the partitioning of binary LAS and LAZ formats, due to the inefficiency problems of the generic text-based ASCII format, which are derived from the reading and interpretation of data and typical large size of files even for small amounts of data. Figure 35 represents a 3D representation of an example cloud point file.

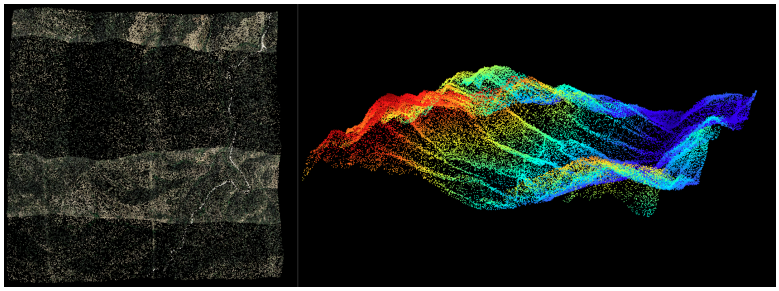


Figure 35: Example cloud point of a mountainous region in the area of Muntanyes de Prades, Tarragona, Spain

LAS is a public file format developed to store LiDAR data in the form of point cloud data. This file format was developed by the American Society for Photogrammetry and Remote Sensing (ASPRS) in 2003 and constitutes one of the main open standard formats used for the storage and processing of LiDAR data. Its intention is to provide an open format that allows different LIDAR hardware and software tools to output data in a common format to facilitate the processing of this type of data. The LAZ file format is a lossless compression of the LAS format.

LAS/LAZ file format consist of 4 differentiated sections:

- Public Header Block: It contains generic metadata such as point number, coordinate bounds or point format.
- Variable Length Records (VLRs): It contains variable types of additional information such as the spatial reference system or descriptions on extra dimensions of the points.
- Point Data Records: It contains the list of points with all its associated data, such as: intensity, classification or some related flags. There exists 10 different types of point data format, each

of them allow storing different fields associated to each point. All Point Data Records in a file must be the of same type.

- Extended variable length records (EVLRs): This section was added in LAS specification version 1.3. The EVLR is, in spirit, identical to a VLR but can carry a larger payload as the "Record Length After Header" field is 8 bytes instead of 2 bytes.

8.2.2 Coordinates-based naive partitioning

To facilitate the parallel processing of LAS/LAZ files, a previous preprocessing of the data is needed in which the files are partitioned.

As detailed above, LAS/LAZ file format consist of different sections, which can basically be divided into two types: metadata and point data. LiDAR file partitioning has to respect the structure of the LAS/LAZ files in order to remain compatible with already existing LAS processing software. To do this, all the metadata of the original file must be kept in each partitioned file and the Point Data Records must be divided between each one, as it is shown in Figure 36.

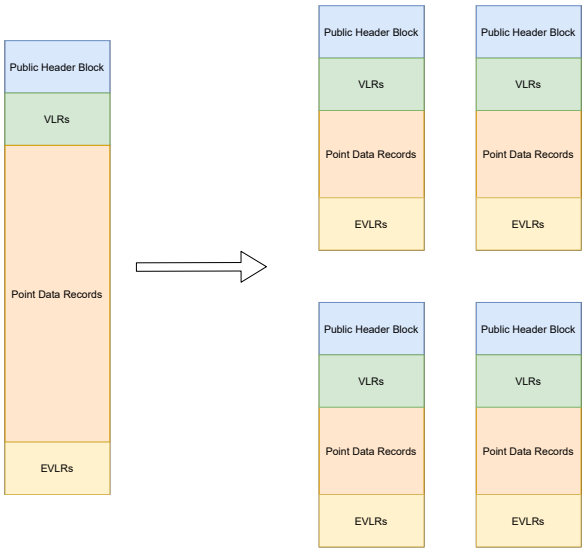


Figure 36: Scheme for LiDAR files partitioning.

A first approach for partitioning a LiDAR file is by chunking the xy-plane in equivalent dimensions. For example, for a lidar file of 1000 square meters, one could partition it into 4 smaller files of 250 square meters each, the first with minimum coordinates $(0, 0)$ and maximum $(250, 250)$, second with $(0, 0)$ minimum and $(250, 500)$ maximum, and so on.

This partitioning method is not optimal. This is mainly due to the fact that LiDAR files can be unbalanced, i.e. the point density is not required to be the same along the entire xy plane of the point cloud. This is due to different reasons. For example, it could be that the file has been formed by different point measurements with different tools or methods (e.g., combining light aircraft stripes with lower altitude drones in specific areas). As a result, partitioning the file from static chunks of the xy plane produces unbalanced partitions, which is not reliable nor optimal for serverless computation: First, performance is affected due to unbalanced load among workers, and second, because some partitions might exceed functions' maximum memory.

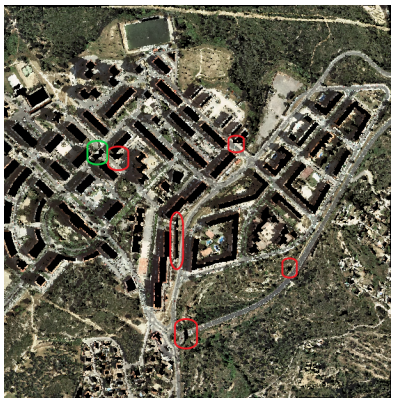
8.2.3 Density-based advanced partitioning

In this section we present the ins-and-outs and the benefits of the LiDAR partitioner we have developed as part of the geospatial toolkit for serverless processing.

In brief, our partitioner has the following benefits over the coordinates-based partitioner described above: (i) we **delimit chunks based on point density instead of by hard coordinate bounds**,

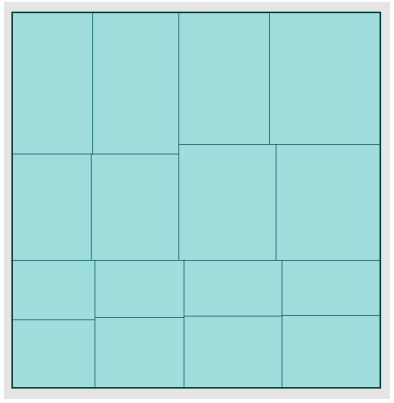
(a) Original point cloud.

(b) Result of SMRF on the entire point cloud.

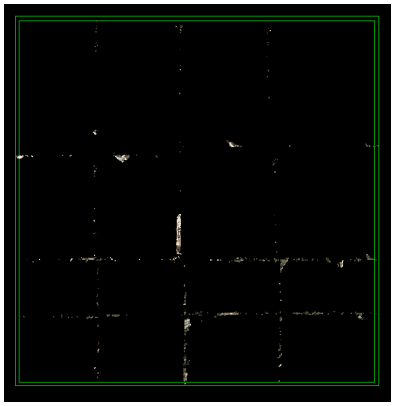


(c) Result of SMRF in on the PointView partitioned in 16 chunks **without** buffer.

(d) Result of SMRF in on the point cloud partitioned in 16 chunks **with** buffer.



(e) Shape of the created tiles without buffer.



(f) The distortions and protrusions problem that arise without buffer.

Figure 37: Example of the problems that arise when the buffer is not used.

which produce even-sized chunks and (ii) we **introduce a buffer to each chunk in order to reduce partition boundary anomalies**.

File granularity size is very important when using serverless functions for processing them: if files are very large, serverless functions will exceed the available function's memory limit; but if the files are very small, computational efficiency will be degraded.

The partitioner has been implemented in Python, using the Numpy and Laspy libraries. The partitioning algorithm is divided into two parts, in the first part it is calculated which tile each point will go to. In the second part, the buffer is computed for each tile.

The computation of the tiles is carried out by means of a recursive function. This function is responsible for dividing the set of points into two tiles, the division is done axially on the widest dimension between x and y. In this way, it is possible to divide the initial set of points into tiles of similar size as square as possible. When the created partitions have the desired size, the recursion stops and returns.

In some types of data processing, rough partitioning can cause tile border anomalies. For example, when we apply a Simple Morphological Filter (SMRF) to a set of points that have been partitioned, we can find anomalies at the border of the tile, since it is an algorithm that depends on neighboring points. A possible solution for this problem (introduced by partitioning) is to add to each tile a buffer with the surrounding points.

Figure 37 shows an example of the anomalies introduced by unbuffered partitioning. In Figure

37b we can see the result of the SFRF applied to the entire file without any previous partitioning. Figures 37c and 37d show the results of the same filter applied to the partitioned file without and with buffer respectively. It can be seen that the result obtained in figure 37d (using the buffer) is the same as the result of the filter applied to the entire file, on the other hand, we see that in figure 37c some anomalies are introduced when no buffer is used. Figures 37e and 37f show us how these anomalies are introduced right at the boundaries of the partitions.

For this reason, the implemented file partitioner allows to add a buffer to each one of the tile partitions.

Once the different tiles have been created, a buffer is added to each one. The size of the buffer is determined by a parameter of the type `float` that sets the maximum distance between the buffer points and the tile. In the buffer computation, for each tile, the minimum and maximum of the dimensions x and y are computed and all the points of the file that meet the following condition are added to the buffer:

$$\{ (x, y) \mid x_{min} - b \leq x \leq x_{max} + b \wedge y_{min} - b \leq y \leq y_{max} + b \}$$

Finally, the points corresponding to the buffer are marked with the `withheld` flag inside the Point Data Records. This allows detecting, when processing the files in parallel, the buffer points in order to quit them from the processing logic. Note that points belonging to the buffer of one tile will be also contained inside another tile. We keep their data to be able to process the border points of the tile, which are adjacent to the buffer points.

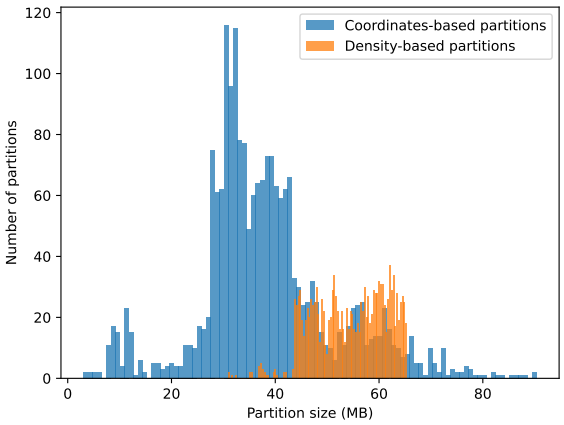


Figure 38: Histogram comparison of coordinates-based histogram and density-based histogram for the Tarragona LiDAR 80GB dataset.

In Figure 38 we can see a histogram comparing the partition sized of both methods: coordinates-based and denisity-based. We can see that using the density-based partitioning, we have much less variability of partition sizes, which proves that the points are more evenly distributed and the workload will be balanced among all functions.

8.2.4 Single-file partitioning performance evaluation

Although the design of the algorithm is sequential, thanks to the use of the NumPy library and the use of vectorized instructions, parallel execution and high computational efficiency are achieved. Below we detail the performance evaluation of the implemented partitioner.

To test and validate the performance of the program, the execution time for the partitioning of files has been measured. We have used five files with different number of points, in particular, the files used contained 1, 2, 4, 8 and 16 million points stored, each file has been partitioned into 10 parts

with a similar size. The measurements have been made on a personal computer that consists of 8 GB of ram memory and a CPU with 4 cores (Intel i5-8250U).

Figure 39 shows the results of the experiments. The partitioner takes 26.5 seconds to chunk a file with 16 million Point Data Records, what gives us an average of 1.65 seconds per million points, a similar measurement to the 1.45 seconds needed to chunk a file with 1 million Point Data Record. Therefore, it can be said that the execution time grows linearly with the number of points to be partitioned. This results proof the efficiency of the developed tool.

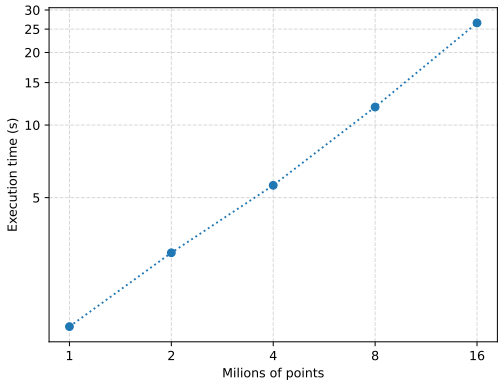


Figure 39: Performance evaluation of the LiDAR partitioner for a single file.

8.2.5 Partitioning of a LiDAR dataset (multiple files)

Figure 40: Area from the city of Tarragona for the 80GB LiDAR dataset.

In this section we will evaluate the performance of Lithops for prerpocessing many LiDAR files using our density-based partitioner. While partitioning a single file is a sequential process, partitioning multiple files can be viewed as a parallel task. Thanks to Lithops, the partitioning task can be mapped across multiple serverless functions in parallel. Since partitioning is a simple task, Serverless Functions remove server management and cost while improving scalability and efficiency.

For this evaluation, a function has been developed that downloads a single LiDAR file, partitions it using the density-based partitioner, and uploads all the partitions to Object Storage. We use Lithops parallel map, which can automatically evaluate the objects stored in a bucket and map each function worker its corresponding file.

We have used a data set consisting of 516 LAZ files ranging from 10MB to 90MB up to a total of 20GB compressed (80GB uncompressed). It is comprised of the city of Tarragona and some of its surroundings. This data set has been extracted from the Spanish National Centre of Geographical

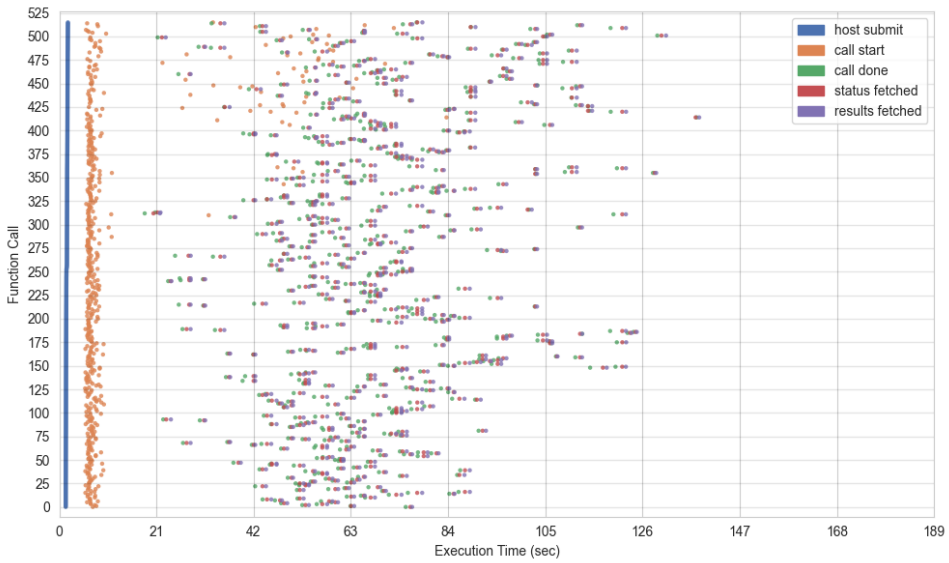


Figure 41: Timeline of the execution of the partitioning pipeline for the 80GB dataset.

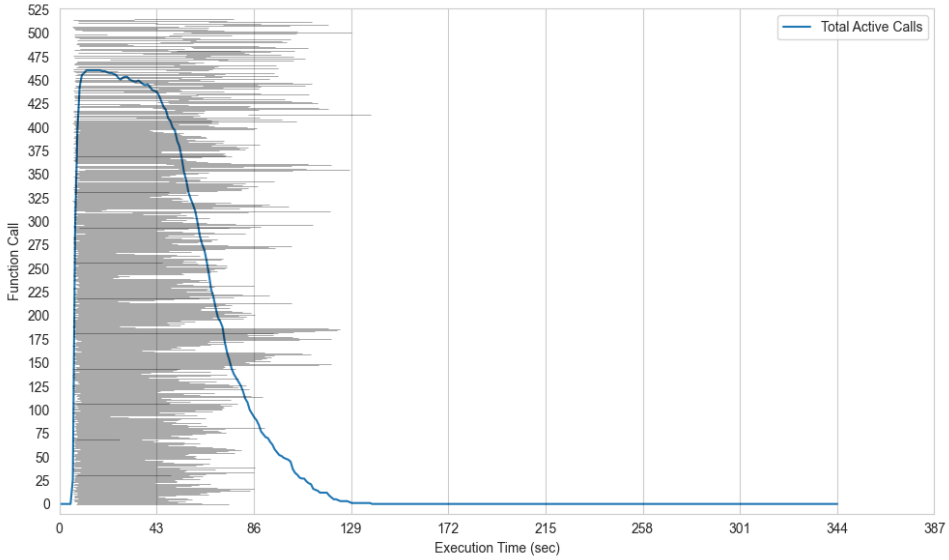


Figure 42: Histogram of the execution of the partitioning pipeline for the 80GB dataset.

Information[2] (Centro Nacional de Información Geográfica, CNIG). In Figure 40 we can see the area which corresponds to this dataset.

Figures 41 and 42 represent the timeline and histogram, respecitvely, of the LiDAR partitioning pipeline for the 80GB dataset. he experiment cost was $0.9911, with a duration of 138.22 seconds and a data-processing throughput of 574.01 MB/s.

8.2.6 Conclusion

We contribute in regard to **simplicity and productivity** since we now provide a free and open source tool for serverless distributes LiDAR data processing. Future data scientists will leverage this tool in order to optimally process point cloud data on the Cloud with minimal effort. Also, we contribute in **performance and scalability**, sice this tool is embedded into Lithops framework, which allows to pre-process and partition many LiDAR files in parallel, e.g. 516 files, 80 GB uncompressed, in 2

[2] `http://centrodedescargas.cnig.es/CentroDescargas/catalogo.do?Serie=LIDAR`.

minutes and 18 seconds.

8.3 Preprocessing Workflow: Geospatial Models Calculation

This workflow consists of preprocessing LiDAR files in order to generate the terrain models needed for other geospatial data streams. In particular, in this workflow LiDAR files are processed in order to generate: Digital Elevation Models (DEM), Digital Surface Models (DSM), Canopy Height Models (CHM), Fraction of Canopy Cover (FCC), Slope and Aspect.

- **Digital Elevation Model (DEM)**: Represents the bare ground (bare earth) topographic surface of the Earth, excluding biological or anthropic elements, for example, trees, buildings, and other surface objects. In this case, we want to ignore any LiDAR classification values that may have already been calculated so that we can derive our own. The PDAL (Point Data Abstraction Library) is a powerful tool for processing LIDAR point cloud.
- **Digital Surface Model (DSM)**: Represents the top of the earth's surface, including biological or anthropic elements, such as trees, buildings and other objects that sit on the earth.
- **Canopy Height Model (CHM)**: Represents the height or residual distance between the ground and the top of the objects above the ground, in other words, is the measurement of the actual heights of trees, buildings, and other objects on the earth's surface. This CHM is obtained by subtracting the DTM from the DSM.
- **Fraction of Canopy Cover (FCC)**: Represents the percentage of the area covered by a vertical projection of the outermost perimeter of tree crowns in each pixel of a raster image.
- **Slope**: Represents the degree of incline of a hill side. The steeper the slope, the faster the fire spreads, and it burns more rapidly uphill than downhill. An explanation for these two phenomena is that the fuel above the fire is brought into closer contact with the upward moving flames. Another concern about steep slopes is the possibility that burning materials roll down the hill and ignite the fuel below the main fire. A surface fire is primarily influenced by the amount of fuel and the wind speed, but a fire starting near the bottom of a slope in normal daytime up slope wind conditions should spread faster and over a larger area than a fire starting near the top of the slope. To sum up, the slope model is a raster image that represents the maximum altitude variation in each pixel of a raster image in relation to the surrounding pixels. The units can be radians, degrees or percentage.
- **Aspect**: Represents the direction a slope is facing. The solar orientation generally determines the amount of heat provided by the Sun and therefore influences the amount, condition, and type of fuel. South southwest slopes are more exposed to sunlight and often correspond to lighter and sparser fuels, higher temperatures, lower humidity, and lower fuel moisture. Consequently, they are most critical in terms of the start and spread of wild land fires. On the contrary, north-facing slopes are less subjected to fire activity than south-facing slopes. They are more shaded, which leads to heavier fuels, lower temperature, higher humidity, and higher fuel moisture. In summary, the aspect model is a raster image that represents the angle that a slope faces with respect to the north. The units can be radians, degrees or percentage.

8.3.1 Data granularity analysis

For this workflow, we have studied the effects on performance of data granularty. In particular, how data partitioning can affect performance.

We want to find out the ideal size of LiDAR files partitions so that we get the lowest execution time. Using our LiDAR partitioner described in Section 8.2 allows to select the required size for each partition. To analyze the most optimal partition size, the workflow has been executed using file sizes of 10 MB, 20 MB, 30 MB, 40 MB, 50 MB and 60 MB. For amount of data, we used two methods: (i)

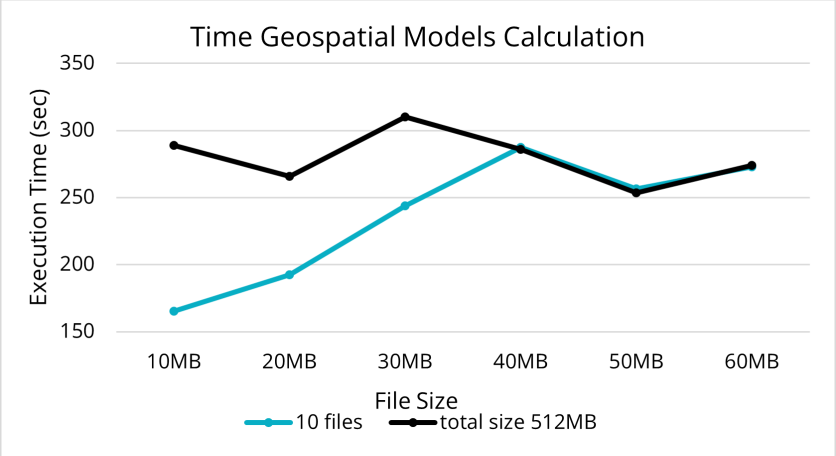


Figure 43: Execution time of Terrain Models Calculation workflow for different file sizes.

execution of ten files of the same size and (ii) execution of a certain amount of files of the same size that in total make approximately 512MB.

The size of the partitions has to be selected considering that it should not be too small to avoid too many calls, but not too large so that the calculation of the terrain models based on the input LiDAR was more distributed among the workers. Figure 43 represents the execution time for different partition sizes. From the results, it has been considered the size of a file 50MB is the most optimal, because it is more efficient than 40MB or 60MB, slightly better than 30 MB and not as small as 10MB or 20MB (for large quantities they would generate many calls).

8.3.2 Partitioning Comparative

In this analysis, we want to measure and compare the workflow performance impact when using raw data (data with no prior pre-processing) and properly partitioned data using the LiDAR partitioner from Section 8.2. The tests have been executed using different total datasets sizes, specifically, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 5 GB and 10 GB.

One reason of performance difference is the number of files to be processed, since each serverless function invocation has overhead, so partitioning into many smaller files can easily decrease performance. It is also the reason for the increase in execution time besides being a completely parallel job with no dependencies.

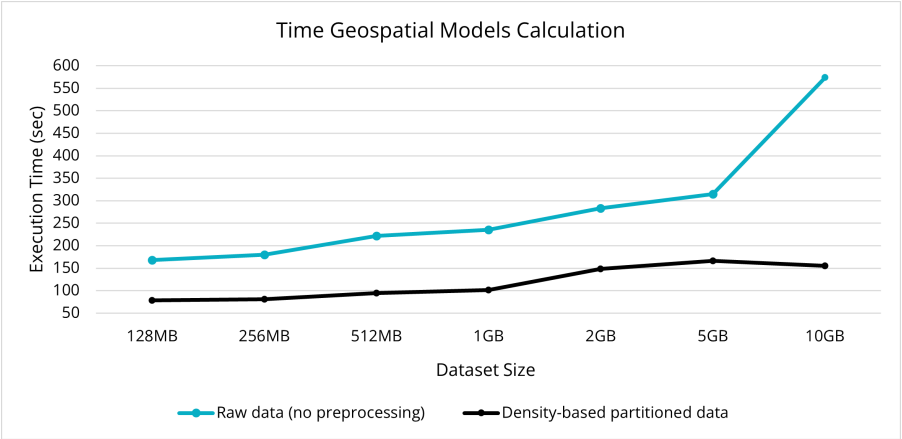


Figure 44: Execution time of Terrain Models Calculation workflow for different file sizes.

Figure 44 depicts the workflow execution time for different dataset sizes comparing no pre-

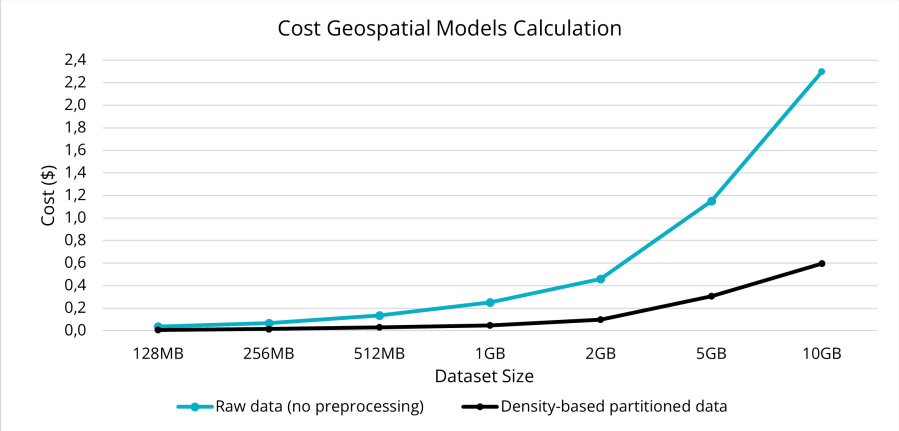


Figure 45: Execution time of Terrain Models Calculation workflow for different file sizes.

processing with balanced partitions. We see that serverless granularity is wasted when raw data is used, since the granularity is bigger and thus, there is less parallelism. On the other hand, thanks to partitioning, the granularity is more fine-grained, allowing to exploit parallelism of serverless and increase speedup. In particular, for the 10GB dataset size, the workflow execution time is 573.44 seconds for raw data and 155.51 seconds for the partitioned data. This is an improvement of $3,68\times$ in terms of execution time. Also, in Figure 45, we can see the cost of the workflow execution for different dataset sizes. As the partitioned dataset version has lower execution time, is much cheaper in comparison with the non-pre-processed dataset version.

8.3.3 Conclusion

In terms of Key Performance Indicators of **simplicity/productivity**, we contribute with the geopastial pipeline for LiDAR data pre-processing which creates Digital Terrain Models necessary for many geospatial applications. Using Lithops and serverless, we can scale and adapt the resources depending on the data to be processed, which also alleviates burden on the data scientists since they don't need to manually manage servers and choose computing capacity. Also, we contribute to **performance/scalability** since, in conjuntion with the serverless LiDAR density-based partitioner described in Section 8.2, we can exploit parallelism of serverless functions to process data with finer granularity. In particular, we can be $3,68\times$ faster when using pre-processing for a 10 GB dataset.

8.4 Preprocessing Workflow: Sentinel2 Satellite Imaging Processing

Data obtained from the ESA Sentinel2 Satellites is raw – meaning that some preprocessing is required prior to usage in other workflows. This section describes the preprocessing workflow used to enable Sentinel2 satellite image to be processed in parallel in many functions.

8.4.1 Sentinel2 satellite images

Sentinel2 is a constellation of two satellites from the European Space Agency Copernicus programme that capture high-resolution images from the Earth's surface [85]. Every five days at the equator, both satellites cover all Earth's land surfaces, large islands, inland and coastal waters, resulting in a huge volume of raw satellite imaging data.

Satellite images are accessed through the Open Data portal Copernicus Open Access Hub. Data is accessible for any user. Images, also called products, are searched using queries, like geographical area, cloud coverage or date. Then, the products are downloaded via HTTP, with a quota of maximum two concurrent downloads per user.

Products are required to be preprocessed and perform **atmospheric correction** prior to being used. Athmospheric correction is a process for satellite images where anomalies in the reflections of the earth's surface caused by external bodies, such as clouds or other satellites, are removed. It is of vital importance to process these images and apply the atmospheric correction, because applying

other processes that are based on the reflectance of the earth's surface can give misleading or incorrect results.

Products retrieved from the Open Data Hub are Level-1C, meaning that they are raw and not processed. Performing atmospheric correction upgrades the product to Level-2A, which can be used in, for example, NDVI calculation. The process to perform athmospheric correction is provided by the Sen2Cor tool included in the Sentinel2 Toolbox [86].

This tool is to what we commonly refer as **black-box**: it is a process that, for a given input, it generates an ouput, and we cannot interfere in the process. The code is not Open Source and, thus, distributed parallelization using serverless functions is not possible. Also, the process is memory and run time demading: the process lasts for 30 minutes in average, which greatly surpasses the limits of serverless functions.

8.4.2 Cloud-Optimized rasters for high-performance parallel processing

Products retrieved from the Open Data Hub are stored in the JPEG 2000 file format. JPEG 2000 allows to define high-resolution satellite image bands (Red, Green, ...) and store multiple bands in the same file, along with added metadata. However, with serverless fine-granularity scaling, we require data formats to be granular and "partition-able" in order to exploit serverless function great parallelism. Unfortunately, JPEG 2000 lacks mechanisms for fine-grained partition of images.

We introduce now The Cloud-Optimized GeoTIFF (COG) data type. COG is defined in [87] as follows:

> A Cloud Optimized GeoTIFF (COG) is a regular GeoTIFF file, aimed at being hosted on a HTTP file server, with an internal organization that enables more efficient workflows on the cloud. It does this by leveraging the ability of clients issuing HTTP GET range requests to ask for just the parts of a file they need.

Cloud-Optimized GeoTIFF incorporates multiple benefits over traditional GeoTIFF raster images. First, data is efficiently retrieved from servers through HTTP Range requests, which enables to consume only a small portion of the data instead of the whole file. Second, it reduces data duplication, since static data chunking in different files is not needed: multiple applications access only one file at a variable granularity. Third, it is backwards compatible with already existing geographical processing software that can read GeoTIFF file formats.

A Cloud-Optimized GeoTIFF differs from a regular GeoTIFF in that the internal pixels of the raster image are re-arranged in multiple square windows instead of in a continuous strip of data, so that applications can leverage HTTP Range requests to retrieve individual windows instead of the whole file. This proves extremely useful for partitioning on the fly a large satellite image and process it in parallel, because **functions can access portions of the image directly from Object Storage instead of creating static partitions with different objects beforehand**. Metadata is embedded into the GeoTIFF file format that describe the number of windows and their offsets.

A part from the athmospheric correction, we also transform the satellite image to Cloud-Optimized GeoTIFF, so that functions can effectively access it in parallel in subsequent workflows (see Section 8.5).

8.4.3 A *ServerMix* approach with Lithops

In short, the conclusion is clear: **serverful resources are needed in order to preprocess Sentinel2 satellite images**. To this end, we leverage Lithops capacity to utilize multiple compute backends while maintaining the same user interface. We have developed multiple serverful backends for multiple clouds (IBM Cloud Virtual Servers and AWS Batch) or for on-premise deployments (Kubernetes). Thanks to Lithops multi-cloud and multi-backend desing, **we can combine serverful and serverless resources in the same workflow**. For this use case, we have used AWS Batch to preprocess Sentinel2 images. AWS Batch allows to allocate more resources (like vCPUs, memory and run time) to functions compared to AWS Lambda. On contrast, the scalability is negatively affected, since task invocation time is much greater (seconds to minutes) compared to FaaS (miliseconds). The reason

being that AWS Batch has to allocate EC2 virtual machines on behalf of the user and schedule tasks depending on the resources requested and the resources available in the compute environment.

Athmospheric correction using Serverful Lithops

Here we will download tile images from Sentinel2 using the previously selected configuration and apply athmospheric correction.

This process is not parallelizable and lasts for over 25 minutes, so it is not suited for serverless functions. We will use Lithops Standalone instead, which uses serverful instances that haven't time limits.

•••

•••

```
[ ]: fexec = lithops.FunctionExecutor(backend='aws_batch', storage='aws_s3', runtime=BATCH_RUNTIME)
     fexec.map(perform_atmospheric_correction, geojson_products["features"])
     combined_keys = fexec.get_result()

[ ]: combined_keys
```

NDVI Computation using Serverless Lithops

Now we will calculate NDVI index of tiles tha thave been downloaded and pre-processed before.

This process can be executed in parallel (for every tile) and in serverless functions.

•••

```
[ ]: fexec = lithops.FunctionExecutor(backend='aws_lambda', storage='aws_s3',
                                      runtime=LAMBDA_RUNTIME, runtime_memory=2048)
     fexec.map(ndvi, combined_keys)
     ndvi_keys = fexec.get_result()
```

Figure 46: Notebook portion where Serverful and Serverless Lithops are combined using in the same notebook maintaining the same API.

Figure 46 shows a portion of the notebook for this preprocessing workflow. We can see how the Lithops API is maintained for both serverful and serverless executions. As an example, NDVI process is used in parallel in the same notebook using serverless functions, in order to demonstrate mixing serverful/serverless resources in the same notebook (more on the NDVI workflow in Section 8.5).

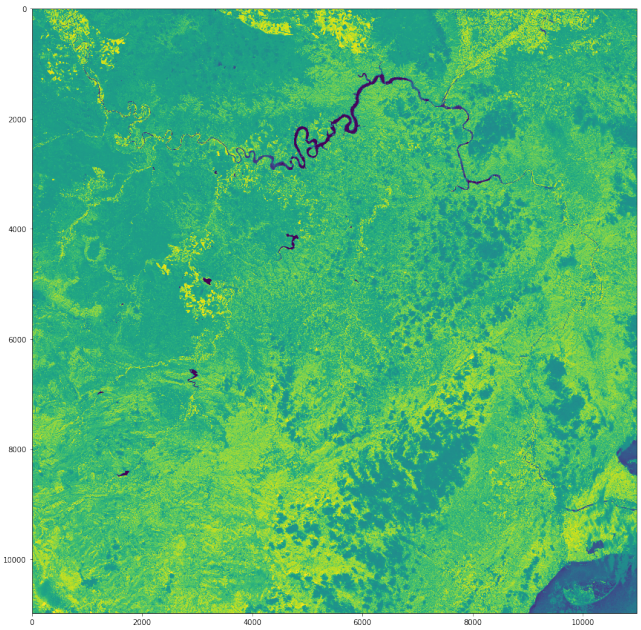


Figure 47: NDVI of the Terra Alta region of Tarragona, Spain.

Figure 47 shows the result of applying NDVI to a product retrieved from the Sentinel2 satellite image dataset.

8.4.4 Conclusion

Refering to the Key Performance Indicators, we can contribute in **simplicity & Productivity** because Lithops enables to combine serverful resources for demanding tasks and serverless resources for highly flexible and parallel tasks. Since the API is maintained, the behaviour is transparent to the user, meaning that the user does not notice of the change in respect of functionality. Also, we contribute in **Performance& Scalability**: thanks to the costly preprocessing of raw Sentinel2 images and converting them to Cloud-Optimized data types, we can later efficiently process that data in parallel and avoid static partitions with many different objects (data duplication).

8.5 Geospatial Workflow: NDVI Calculation

In this section, we will describe the NDVI workflow. NDVI means "normalized difference vegetation index", and it's an indicator that identifies whether the target being studied contains live green vegetation or not [88].

8.5.1 Normalized Difference Vegetation Index

Chlorophyll from live green plants absorb solar radiation to be used as a source of energy in the process of photosynthesis. Leaf cells re-emit solar radiation in the near-infrared spectral region because the photon energy at infrared wavelengths are too large to synthesize organic molecules, also because it would overheat the cells and cause tissue damage. The cell structure of the leaves, on the other hand, strongly reflects near-infrared light. Therefore, we can calculate the ratio of reflectance comparing the Red band and NIR (Near-Infrared) bands of a raster image:

$$NDVI = \frac{(NIR - Red)}{(NIR + Red)}$$

NDVI is useful to assess the vegetation of an area to identify potential risk of drought or wildfire risk. In this workflow we will calculate NDVI of an area of two moments in time and calculate the difference, which will provide insight on the areas where vegetation is degrading faster, in order to take action and possibly avoid a wildfire risk.

Although NDVI computation is fairly simple, we want to emphasize that the computational complexity is not as important as the volume of data taken into account. Computing the NDVI of a small forest area, for example, is fast; but compute complexity scales linearly for multiple dates, for time lapses of NDVI evolution, or for big regions, nation-wide areas of terrain.

8.5.2 Cloud-Optimized GeoTIFFs and the AWS Open Data Registry

This workflow is prepared to accept and consume the data produced in preprocessing workflow using Sentinel2 images (see Section 8.4). In this workflow we will utilize, however, a public dataset from Amazon Web Servicies Registry of Open Data, where Sentinel2 images are already prepared to be utilized by applications [89].

Data is stored using the Cloud-Optimized GeoTIFF data format [87]. As explained in Section 8.4, a Cloud-Optimized GeoTIFF file has its data arranged in multiple windows that can be accessed on demand and by any granularity using HTTP Range requests. This feature enables to partition on-the-fly Cloud-Optimized GeoTIFF files stored in Cloud Object Storage by many functions in parallel, avoiding costly processes of static chunking with multiple objects, which leads to data duplication.

8.5.3 Sample execution: NDVI difference using Lithops

By processing multiple Cloud-Optimized GeoTIFF windows in parallel, we greatly exploit massive parallelism of serverless functions. We present now a sample execution of the NDVI workflow where the difference of NDVI comparing two different dates from multiple regions of the San Francisco area, California, USA. In Figure 48 we can see the surface area that is going to be processed.

In total, there are 8 tiles arranged in 11×11 windows, which add up to 968 windwos to be computed. That is, 968 serverless functions running in parallel. The total dataset is 5.44 GB in size.

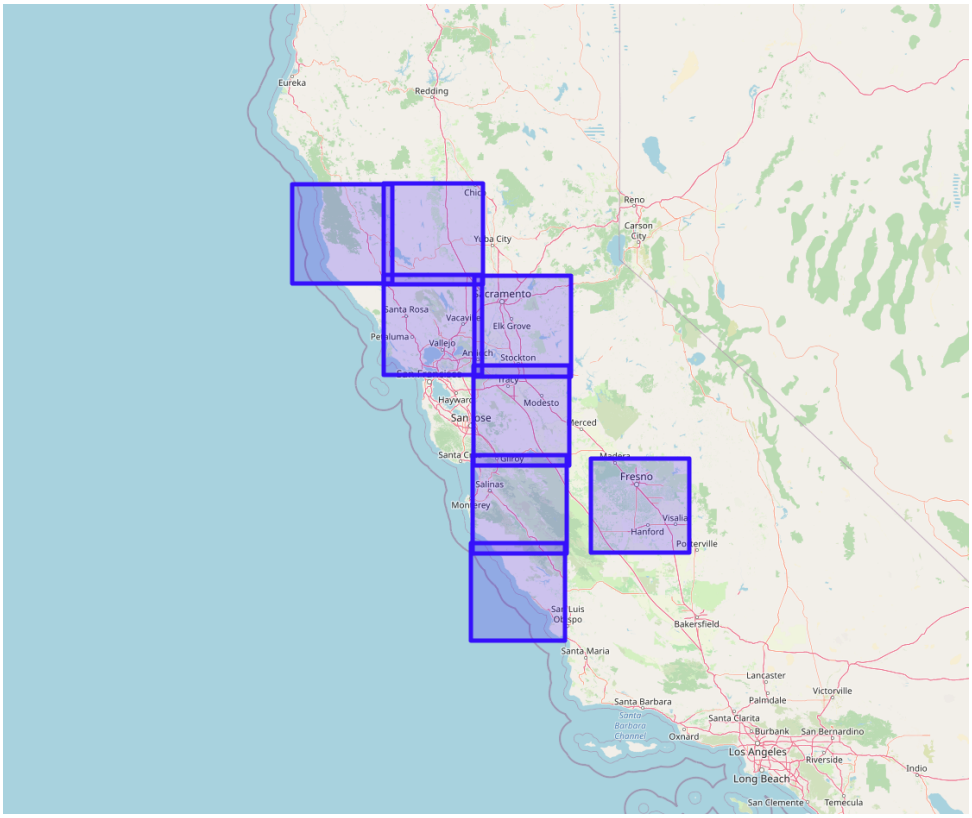


Figure 48: Processed areas for the NDVI workflow of the San Francisco area, California, USA.

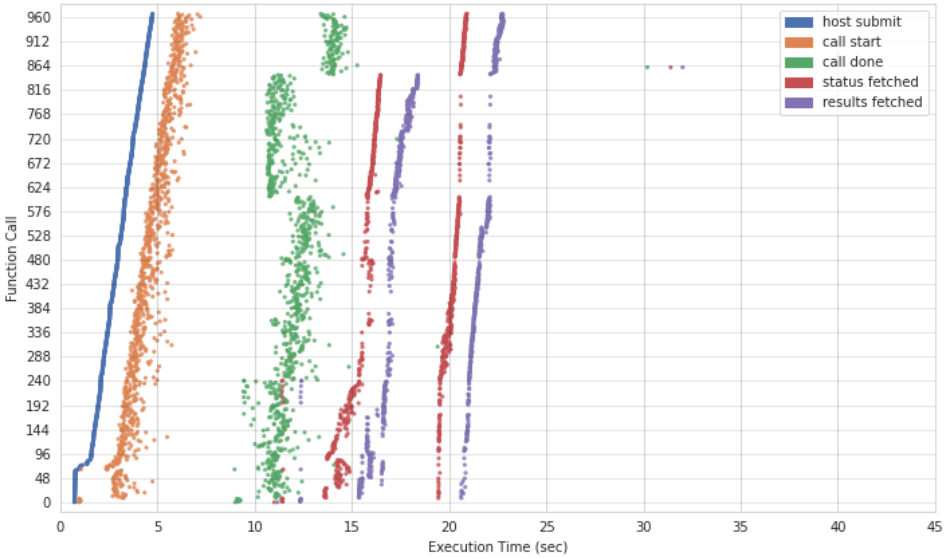


Figure 49: Timeline of the workflow sample execution.

Figures 49 and 50 represent a timeline and histogram of the workflow execution, respectively. We see that all 986 functions are being run in parallel, which proves high scalability and parallelism of serverless functions. The overall function run time is $\approx 4.37s$, which implies to a throughput of **1.25 GB/s** of data processed. Object Storage allows to read concurrently from many objects in a shared-nothing architecture, which entails huge concurrent read throughput. The whole sample execution cost is 0.11 USD.

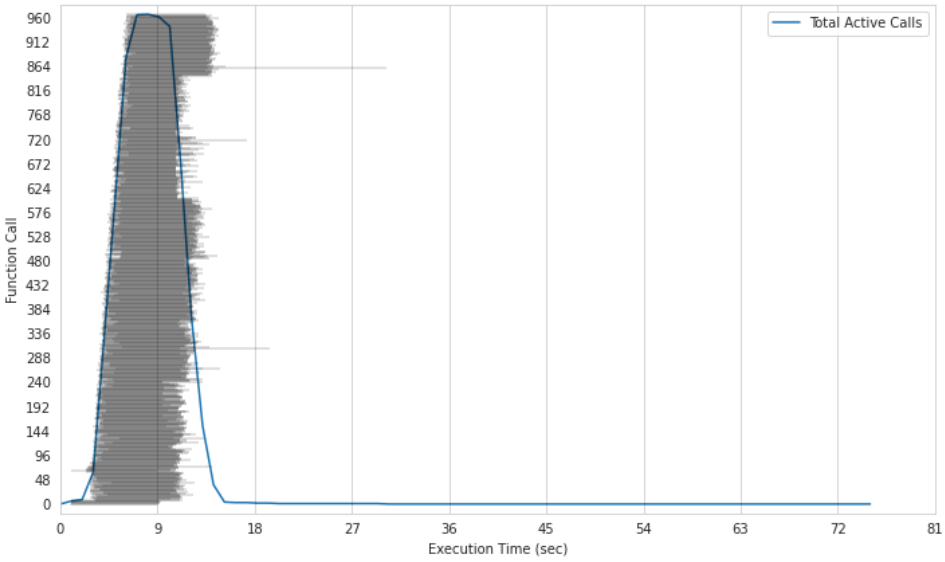


Figure 50: Histogram of the workflow sample execution.

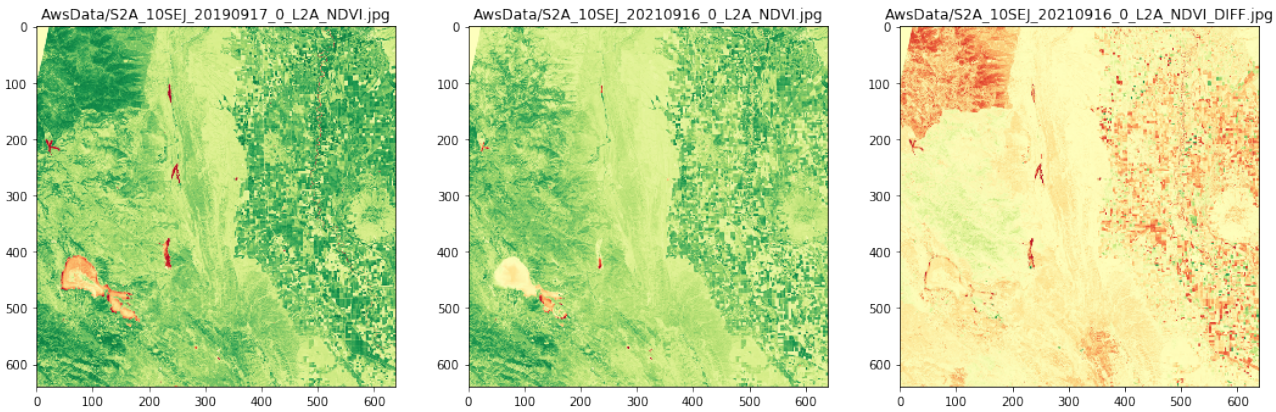


Figure 51: Result of the sample NDVI workflow execution.

Figure 51 represents the result of a tile from the sample workflow execution. We can see first both NDVI rasters from different dates. The third raster represents the difference between the two. With this raster we can easily identify forest areas where vegetation has decreased (for example, by wildfires or deforestation) throughout the time window.

8.5.4 Conclusion

As a conclusion, we will refer to the Key Performance Indicators obtained from this workflow. First, we have contributed with **simplicity/productivity** since we now provide a tool that facilitates parallel processing of many Cloud-Optimized GeoTIFF files sotred in Cloud Object Storage. The data scientist will no longer need to manually manage partitions since Lithops does it automatically with the use of this tool. Also, we contribute to **performance/scalability** with the usage of serverless functions for massive parallel computations. Together with the partitioning tool for Cloud-Optimized GeoTIFFs, we provide a powerful toolset to process geospatial data at scale in the Cloud at low monetary and management cost using serverless technologies.

8.6 Geospatial Workflow: Water Consumption

This workflow consists on comparing water use estimates obtained from two different databases over extended regions of irrigated crop fields and then map differences in the water use footprint of irrigated arable lands in representative large areas of Peninsular Spain.

On the one hand, we use high-resolution NDVI index (Normalized Difference Vegetation Index) derived from satellite imagery obtained from workflow NDVI Calculation (see Section 8.5), we identify actual irrigated crop areas and we estimate water consumption using multi-date imagery data along the growing season. The continuous update of open-access databases and the utilization of the CloudButton Toolkit capabilities make possible the mapping these variations along a certain period of time with frequent updates.

On the other hand, we estimate and map water consumption indicators considering the officially declared and georeferenced irrigated arable land area which is available from SIGPAC, the Geographic Information System for the Agricultural Common Policy open access database and specific correction factors (irrigated land area where to calculate the crop water consumption volume).

The comparison of both results identifies non-coincident areas which help to monitor water use efficiency and funding resource allocation.

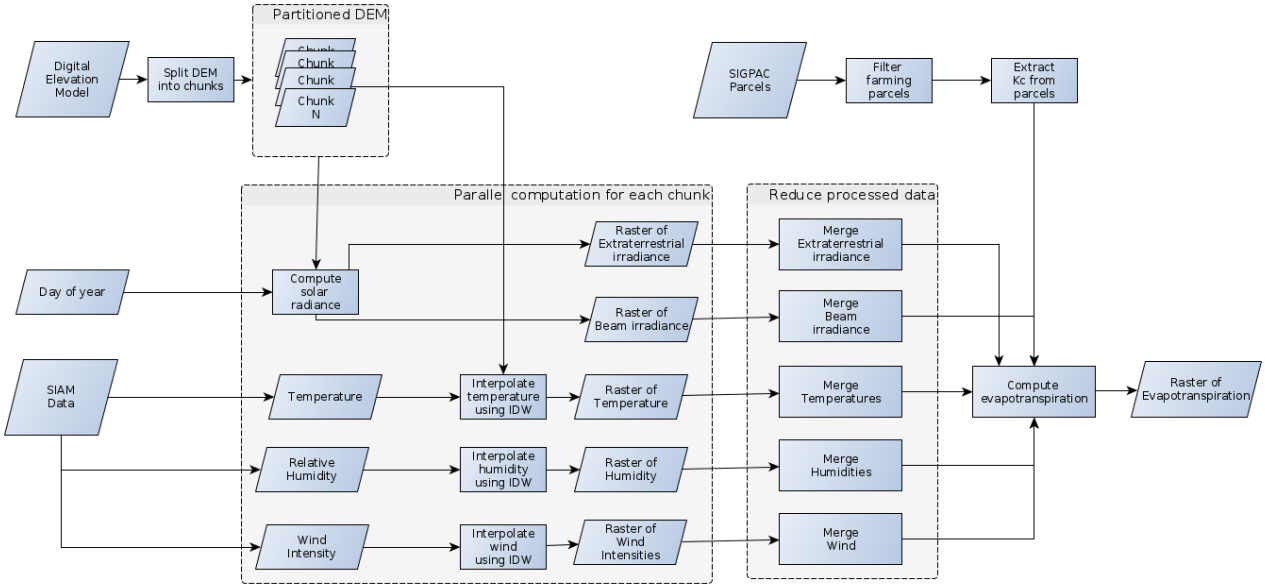


Figure 52: Diagram of the Water Consumption workflow steps, data and dependencies.

In a nutshell, the workflow consists of the following steps. First, the input Digital Terrain Models are retrieved from Object Storage and pre-processed and partitioned accordingly. We partition the each tile in 4 chunks, in order to exploit serverless functions parallelism, increase speedup and reduce overall execution time. For each chunk, a interpolation raster of radiance, extraterrestrial radiance, wind, humidity and temperature are computed using metadata retrieved from the SIAM platform, which contains point data of weather stations across the region. Then, each chunk is merged and the original tile size is recomposed, one for each kind of interpolation raster. Finally, for each, tile, we compute evapotranspiration using the Penman-Monteith formula and by combining all interpolation rasters with added crop metadata by land plot.

In Figure 53 we can see the result from a tile after applying the Water Consumption workflow. The result raster contains information of the crops fields and calculated water consumption.

8.6.1 Workflow execution sample

In this section, we describe a sample execution of the Water Consumption workflow. We have used data obtained from the Centro Nacional de Información Geográfica (National Centre of Geographical

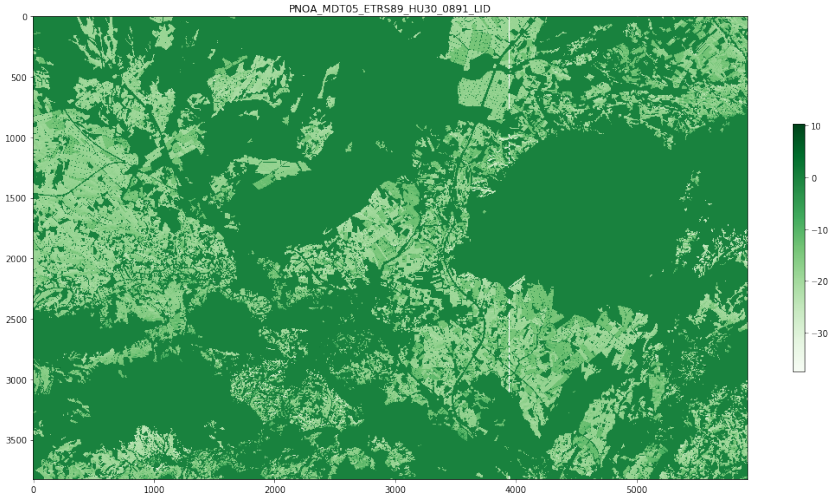


Figure 53: Result tile of Water Consumption workflow.

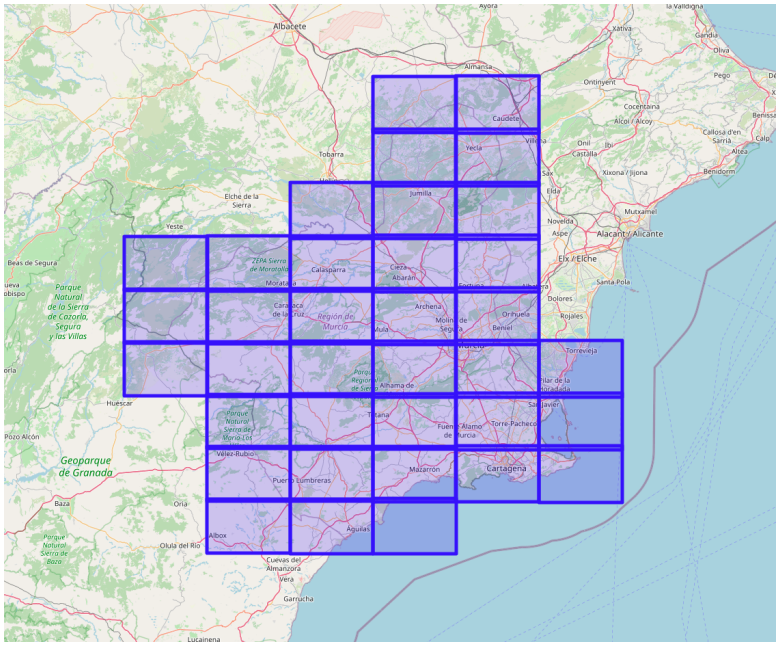


Figure 54: Land area processed by the Water Consumption workflow execution sample.

Information) to process the whole autonomous community of Región de Murcia in Spain. The input dataset consists of 36 objects of Digital Terrain Models covering the whole area, totaling 6.07 GB which cover 11.313 square kilometers of surface area. In Figure 54 we can see the 36 tiles covering the surface area to be processed.

Table 3 displays the execution statistics of the sample execution. We can see that we use a total of 1908 functions, with 2048 MB of memory each. The longest workflow step is `radiation_interpolation` with an average time of 150 seconds. The rest of the workflow steps are relatively fast thanks in part of partitioning in smaller chunks and exploiting parallelism. The total workflow execution has cost 1.85 USD for 6.07 GB of data processed.

Figures 55 and 56 depict a timeline and histogram, respectively, of the workflow sample execution. We can see that the maximum parallelism peak is reached at 1296 parallel functions, which

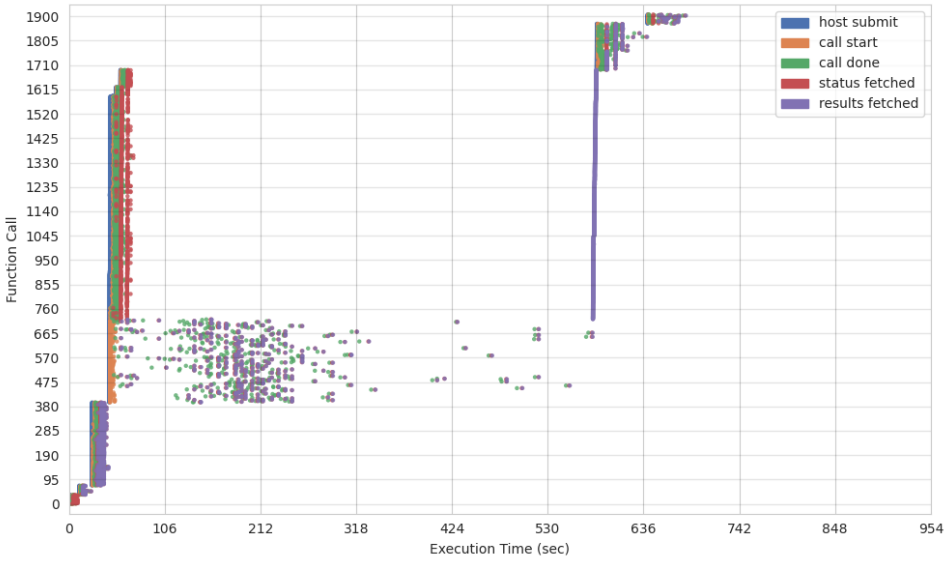


Figure 55: Timeline of the Water Consumption workflow execution sample using Lithops.

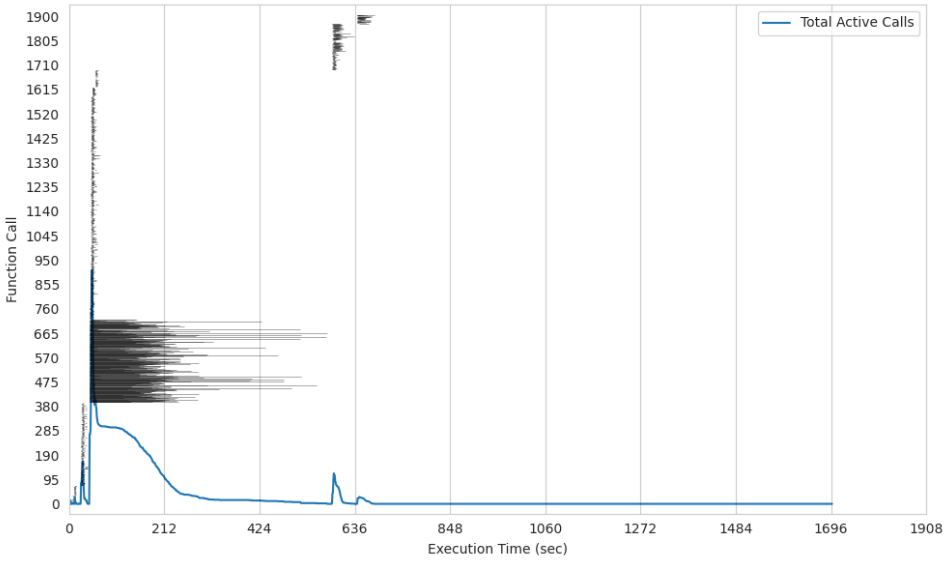


Figure 56: Histogram of the Water Consumption workflow execution sample using Lithops.

corresponds to all combined functions from interpolation calculation steps. We see that the workflow execution is very heterogeneous in terms of task execution time and granularity, and that it changes fast and sharply across the workflow, meaning that resources needed is not steady throughout the workflow. However, thanks to flexible resource allocation of serverless functions using Lithops, we can always adapt to the perfect resource utilization across the whole workflow execution. Other approaches using static serverful cluster computing resources do not have this instant flexibility, causing over-provisioning (waste of money) or under-provisioning (waste of compute capacity).

8.6.2 Conclusion

With this workflow, we demonstrated that, with serverless computing with Lithops, we can improve **performance/scalability** because the compute resources are always adapted at a very fine granularity, but also in **simplicity/productivity** since the data analyst has no longer to worry about server management or resource allocation. Both combined provide *always-right* allocation of resources,

Job ID	Function	Invocations	Memory (MB)	Avg Run time (s)	Cost (USD)
M000	asc_to_geotiff	36	73728	0.85	0.001
M001	get_tile_meta	36	73728	1.27	0.001
M002	split_blocks	324	663552	2.63	0.029
M003	radiation_interpolation	324	663552	149.70	1.649
M004	temperature_interpolation	324	663552	3.35	0.036
M005	humidity_interpolation	324	663552	3.12	0.034
M006	wind_interpolation	324	663552	3.12	0.034
M007	merge_blocks	180	368640	8.33	0.051
M008	combine_calculations	36	73728	14.46	0.017
Summary		**1908**	**3907584 MB**	**28.60 s**	**1.85 $**

Table 3: Execution statistics for Water Consumption workflow.

which minimizes waste of compute power but also maximizes resource usage for higher parallelism.

8.7 Geospatial Workflow: Biomass Calculation

In this workflow, we will calculate the biomass of a delimited area. We will be using the Canopy Height Model discrete LiDAR data product as well as CNIG (Spanish National Geographic Information Center) field data on vegetation data. This process will calculate biomass for individual trees in a forest. This workflow has been adapted and extended from NeonScience Learning Hub [90] for Lithops and parallel distributed computing.

The calculation of biomass consists of four primary steps: First, we delineate individual tree crowns, then we calculate predictor variables for all individuals. After that, we collecting training data and finally we apply a regression model to estimate biomass based from predictors.

Before entering the actual calculations we first are going to explain the two main file formats used in this pipeline:

- The **ASCII Raster File** format is a simple format that can be used to transfer raster data between various applications. It is basically a few lines of header data followed by lists of cell values. In this case, we will download maps (in .asc format) from the land we want to work on.
- **GeoTIFF** files are raster image file types that are commonly used to store satellite and aerial imagery data, along with geographic metadata that describes the location in space of the image.

8.7.1 Filtered Canopy Height Models

The data obtained from the data source is in ASCII format. We convert this file to a cloud-optimized GeoTIFF to then split it in as many smaller tiles as we want to make all calculations, in order to exploit parallelism.

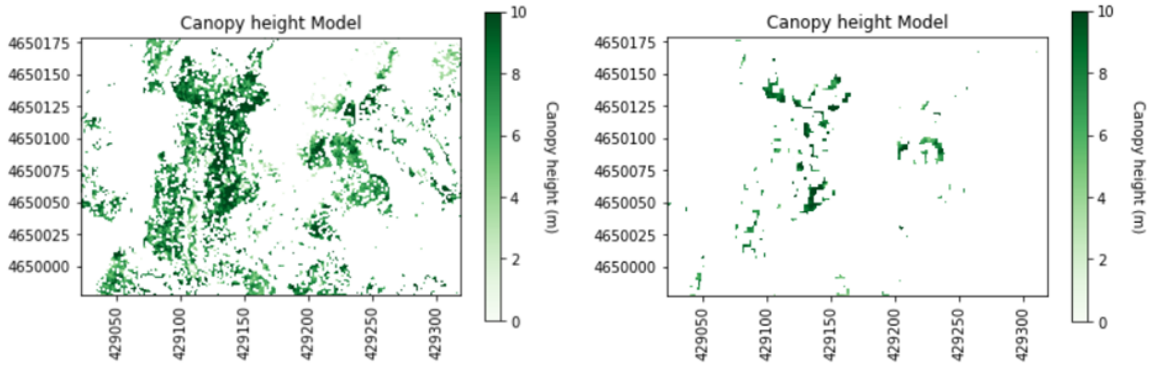


Figure 57: CHM models, raw at left, filtered at right.

We use a Gaussian smoothing kernel (convolution) across the model to remove spurious high vegetation points. This will help ensure we are finding the treetops properly before running a watershed segmentation algorithm. In Figure 57 we see the Canopy Height Model data to be processed. At the left, we see the raw data from source, and at the right, the filtered data.

8.7.2 Determine local maximums and watershed segmentation

The next step is to to determine local maximums within the image. The footprint parameter is an area where only a single peak can be found. This should be approximately the size of the smallest tree. Once we run the algorithm we obtain an array which identifies each pixel being the tree tops. As opposed to the rest of the workflow satges, this stage in particular is computationally expensive. We will target this stage to parallelize and obtain better execution time and speedup.

Next, we will perform the watershed segmentation which produces a raster of labels. After doing that, we will get several properties of the individual trees will be used as predictor variables and then we will get the predictor variables to match the training data that we will use in the next stage.

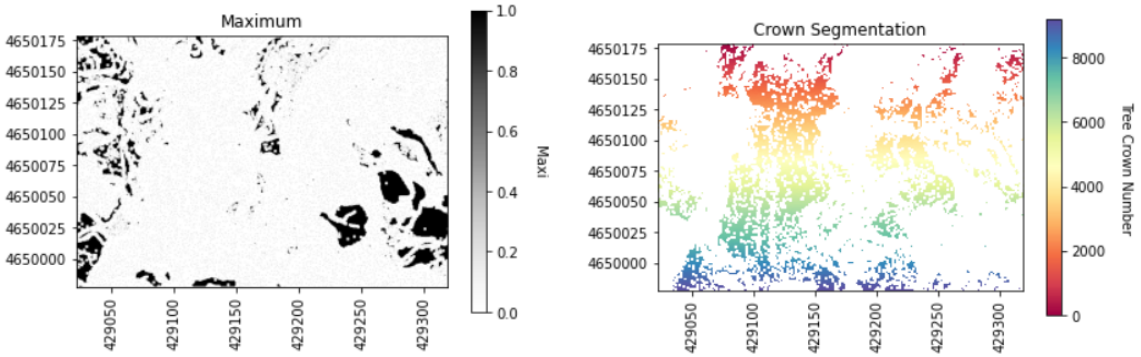


Figure 58: Local maximums plot at left and watershed crown segmentation at right.

Figure 58 displays the local maximums and watershed segmentation outputs of this step.

8.7.3 Training data and Random Forest classification

The training data is a simple CSV file which contains data about biomass and the predictor variables for tree classification. The tree diameter and max height are defined in the NEON vegetation structure data along with the tree DBH. The field validated values are used for training, while the other were determined from the CHM and camera images by manually delineating the tree crowns and pulling out the relevant information from the CHM. Biomass was calculated from DBH according to the formulas in [91].

We use Random Forest classifier and fit the predictor variables from the training data to the Biomass estimates. Finally, we apply the Random Forest model to the predictor variables to retrieve biomass.

Figure 59 depicts the result plot that contains the biomass data for the selected study area.

8.7.4 SpeedUp & Parallelism

In this section we will talk about what we have done to the pipeline to improve it's performance and reduce it's execution time. As mentioned above, it has been achieved by applying parallelism at the local maximum stage, which is computational expensive and can be easily parallelizable. To achieve parallelism, we split the input data array in many chunks, process them in parallel using Lithops and serverless functions, and then merge them together again in order to continue with the workflow.

We used a 600 MB input CHM file. Although 600 MB is relatively a small volume of data, the sequential version of finding local maximums run for 4 hours, 21 minutes y 3 seconds (15663 seconds) using the user's laptop. We tested many splits in order to find maximum speedup. In particular, 4, 5, 10, 15, and 20 splits, which correspond to 16, 25, 100, 225 and 400 parallel functions respectively.

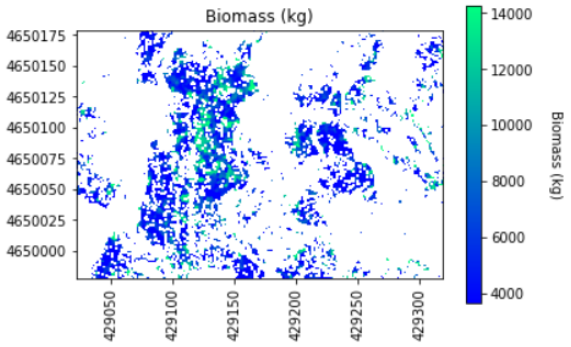


Figure 59: Biomass plot for selected study area.

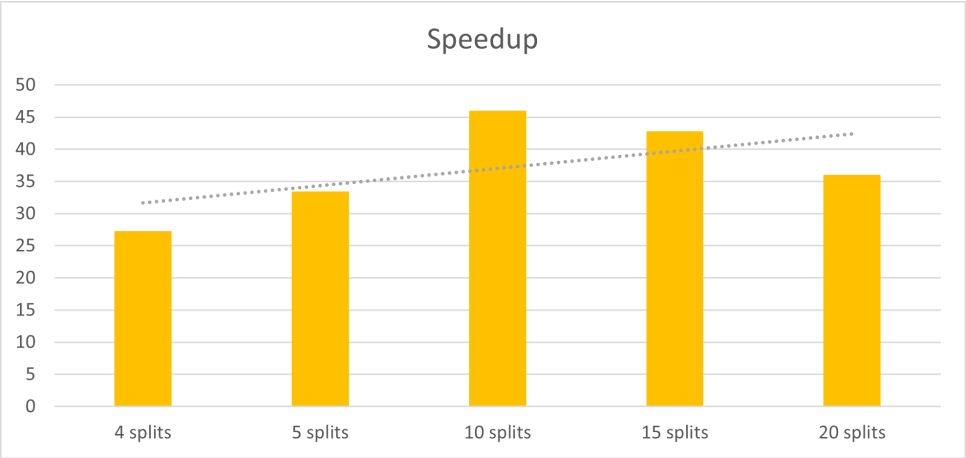


Figure 60: Speedup of the workflow execution by splitting the input image in many chunks to increase parallelism.

In Figure 60 we can see the speedup for all splits. We see that for 20 splits, the workflow stage run for 341.07 seconds, which translates to a speedup of 46×.

Finally, and talking about KPIs, we can say that we are able to achieve a great improvements in **simplicity/productivity** because we can easily adapt existing code and paralellize compute-intensive tasks using Lithops. Also, in **performance/scalability**, allowing to get speedup of up to 46× at zero management cost.

9 Conclusions

In this deliverable, we present the final reference implementation of the CloudButton toolkit (Lithops). We have met the main KPIs of this project: 1. *Simplicity & Productivity* and 2. *Performance & Scalability & Elasticity*, using three use cases with massive (un)structured data: *genomics*, *metabolomics*, and *geospatial* data.

After three years, we have demonstrated that Serverless Data Analytics is a reality with a bright future. In fact, all major Cloud providers offer today Serverless Analytics services such as Amazon (RedShift, MSK, EMR), Google (Serverless Spark) and IBM (IBM Analytics Engine, Lithops). If in 2019 Serverless Data Analytics was a research area, today in 2022, it is a hot area populated by the major Cloud providers. The problem is that we still have a severe vendor-locking problem that ties users to specific deployments.

To mitigate vendor lock-in, Lithops is today a mature multi-cloud software toolkit that is used in production in different deployments (EMBL Metaspace, IBM Finance, URV Metabolomics platform). In this deliverable, we have demonstrated with clear KPIs how Lithops can process massive data in parallel from three domains. Finally, we have created data partitioning and management libraries that considerably simplify *Serverless Genomics*, *Serverless Metabolomics*, and *Serverless Geospatial*.

Lithops is a consolidated project that will continue its progress after the end of CloudButton. In particular, IBM is using it, but also two startups that have been founded by CloudButton partners: EMBL's SpaceM and URV's DATOMA. Serverless Computing is becoming a consolidated trend in all public Clouds, and we foresee further technology advances that may make Serverless Data Analytics even more popular in the next years.

References

[1] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, SoCC'17, 2017.

[2] P. García-López, A. Slominski, S. Shillaker, M. Behrendt, and B. Metzler, "Serverless end game: Disaggregation enabling transparency," arXiv preprint arXiv:2006.01251, 2020.

[3] J. Sampé, P. Garcia-Lopez, M. Sánchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona, "Toward multicloud access transparency in serverless computing," IEEE Software, vol. 38, no. 1, pp. 68–74, 2020.

[4] D. Barcelona-Pons and P. García-López, "Benchmarking parallelism in faas platforms," Future Generation Computer Systems, vol. 124, pp. 268–284, 2021.

[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17), 2017.

[6] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," arXiv preprint arXiv:1812.03651, 2018.

[7] E. J. et al, "Cloud programming simplified: A berkeley view on serverless computing," https://arxiv.org/abs/1902.03383, 2019.

[8] CloudButton Consortium, "Deliverable D2.1 - Experiments and Initial Specifications."

[9] J. Sampe, M. Sanchez-Artigas, P. Garcia Lopez, and G. Paris, "Data-driven serverless functions for object storage," in Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Middleware '17, (New York, NY, USA), pp. 121–133, ACM, 2017.

[10] Y. Kim and J. Lin, "Serverless data analytics with Flint," CoRR, vol. abs/1803.06354, 2018.

[11] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," 2018.

[12] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), (Boston, MA), pp. 193–206, USENIX Association, 2019.

[13] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in 2019 USENIX Annual Technical Conference (ATC'19), pp. 475–488, 2019.

[14] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 923–935, 2018.

[15] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), (Carlsbad, CA), pp. 427–444, USENIX Association, 2018.

[16] D. B. Pons, P. G. López, Á. R. Ollobarren, A. Gómez-Gómez, G. París, and M. S. Artigas, "Faas orchestration of parallel workloads," in Proceedings of the 5th International Workshop on Serverless Computing, WOSC@Middleware 2019, Davis, CA, USA, December 09-13, 2019, pp. 25–30, ACM, 2019.

[17] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto, "Comparison of faas orchestration systems," in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 148–153, IEEE, 2018.

[18] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, Distributed Systems: Concepts and Design. USA: Addison-Wesley Publishing Company, 5th ed., 2011.

[19] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on diåstributed computing," in International Workshop on Mobile Object Systems, pp. 49–64, Springer, 1996.

[20] T. Wagner, "The Serverless SuperComputer," 2019.

[21] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency.," in HotOS, vol. 13, pp. 11–11, 2011.

[22] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," Communications of the ACM, vol. 60, no. 4, pp. 48–54, 2017.

[23] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 1–16, 2019.

[24] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2p2: Making rpcs first-class datacenter citizens," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp. 863–880, 2019.

[25] C. Lee and J. Ousterhout, "Granular computing," in Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '19, p. 149–154, 2019.

[26] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 249–264, 2016.

[27] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed OS for hardware resource disaggregation," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 69–87, 2018.

[28] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," ACM Transactions on Computer Systems (TOCS), vol. 33, no. 4, pp. 1–30, 2015.

[29] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?," in 15th Workshop on Hot Topics in Operating Systems (HotOS 15), 2015.

[30] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 649–667, 2017.

[31] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS track: Building stateful distributed applications with serverless architectures," in Proceedings of the 20th International Middleware Conference, pp. 41–54, 2019.

[32] J. Zhi, R. Wang, J. Clune, and K. O. Stanley, "Fiber: A platform for efficient development and distributed training for reinforcement learning and population-based methods," arXiv preprint arXiv:2003.11164, 2020.

[33] K. Jayaram, V. Muthusamy, P. Dube, V. Ishakian, C. Wang, B. Herta, S. Boag, D. Arroyo, A. Tantawi, A. Verma, et al., "Ffdl: A flexible multi-tenant deep learning platform," in Proceedings of the 20th International Middleware Conference, pp. 82–95, 2019.

[34] S. Shillaker and P. Pietzutch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in 2020 USENIX Annual Technical Conference (USENIX ATC 19), 2020.

[35] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017, pp. 185–200, 2017.

[36] WebAssembly, "WASM Module Specification," 2020.

[37] Mozilla, "WASI: WebAssembly System Interface," 2020.

[38] A. Arjona, G. Finol, and P. Garcia-Lopez, "Transparent serverless execution of python multiprocessing applications," 2022.

[39] "Redis." `https://redis.io/`.

[40] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Disk-locality in datacenter computing considered irrelevant.," in HotOS, vol. 13, pp. 12–12, 2011.

[41] L. Liu, W. Cao, S. Sahin, Q. Zhang, J. Bae, and Y. Wu, "Memory disaggregation: Research problems and opportunities," in 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1664–1673, IEEE, 2019.

[42] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pp. 401–414, 2014.

[43] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al., "The case for ramcloud," Communications of the ACM, vol. 54, no. 7, pp. 121–130, 2011.

[44] Y. Huang, X. Yan, G. Jiang, T. Jin, J. Cheng, A. Xu, Z. Liu, and S. Tu, "Tangram: bridging immutable and mutable abstractions for distributed data analytics," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp. 191–206, 2019.

[45] RISELab, "Apache Ray." https://github.com/ray-project/ray.

[46] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of temporary storage in the nodekernel architecture," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp. 767–782, 2019.

[47] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," Nature Nanotechnology, pp. 1–16, 2020.

[48] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in USENIX Annual Technical Conference, 2018.

[49] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), 2020.

[50] V. Sreekanti, C. W. X. C. Lin, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," arXiv preprint arXiv:2001.04592, 2020.

[51] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "Micro" Back in Microservice," USENIX Annual Technical Conference (USENIX ATC), 2018.

[52] Microsoft Research, "Krustlet."

[53] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," in Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, HotOS '19, ACM, 2019.

[54] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in ACM international symposium on New ideas, new paradigms, and reflections on programming & software, 2013.

[55] D. Buchaca, J. Marcual, J. L. Berral, and D. Carrera, "Sequence-to-sequence models for workload interference prediction on batch processing datacenters," Future Generation Computer Systems, 2020.

[56] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pp. 589–603, 2015.

[57] M. Brooker, T. Chen, and F. Ping, "Millions of tiny databases," in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 463–478, 2020.

[58] Github, "Smile (statistical machine intelligence and learning engine)." https://github.com/otrack/smile.

[59] Contributors of PyWren for IBM Cloud, "PyWren on Knative." `https://github.com/pywren/pywren-ibm-cloud/blob/master/docs/knative.md`, 2020.

[60] LSDS Group, "Faasm Kubernetes/ Knative integration." `https://github.com/lsds/faasm/blob/master/docs/kubernetes.md`, 2020.

[61] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," in Proceedings of the 14th Annual ACM Conference on Distributed Event Based Systems, DEBS '20, 2020.

[62] "Apache Airflow." `https://github.com/apache/airflow`, 2018.

[63] CloudButton Consortium, "Deliverable D4.2 - Specification and partial support for degradable objects."

[64] CloudButton Consortium, "Deliverable D5.2 - CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools."

[65] Amazon, "AWS Lamda." `https://docs.aws.amazon.com/lambda/`.

[66] IBM, "Cloud Functions." `https://cloud.ibm.com/docs/openwhisk`.

[67] Amazon, "AWS Fargate." `https://aws.amazon.com/fargate/`, 2017.

[68] Google, "Google Cloud Run." `https://cloud.google.com/run/`, 2019.

[69] "Knative Platform." `https://cloud.google.com/knative/`, 2019.

[70] KEDA, "Kubernetes-based event-driven autoscaling." `https://keda.sh/`.

[71] E. N. Archive, "Statistics." `https://www.ebi.ac.uk/ena/browser/about/statistics`, 2022.

[72] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," Nature biotechnology, vol. 35, no. 4, pp. 316–319, 2017.

[73] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," Nature methods, vol. 9, no. 12, pp. 1185–1188, 2012.

[74] L. Ferretti, C. Tennakoon, A. Silesian, G. Freimanis, and P. Ribeca, "Sinple: Fast and sensitive variant calling for deep sequencing data," Genes, 2019.

[75] "OCaml website." `https://ocaml.org`, 2020.

[76] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in USENIX Annual Technical Conference (USENIX ATC), 2020.

[77] LSDS Group, "Faasm Python Support." `https://github.com/faasm/python`, 2020.

[78] LSDS Group, "Faasm C/C++ Support." `https://github.com/faasm/cpp`, 2020.

[79] S. Marco-Sola, M. Sammeth, R. Guigó, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," Nature Methods, 2012.

[80] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," Nucleic acids research, vol. 38, no. 6, pp. 1767–1771, 2010.

[81] Luc van Donkersgoed, "Aws re:invent 2020 day 3: Optimizing lambda cost with multi-threading." `https://www.sentiatechblog.com/aws-re-invent-2020-day-3-optimizing-lambda-cost-with-multi-threading`, 2022.

[82] IBM, "IBM Topics: What is geospatial data?," 2022.

[83] T. A. S. for Photogrammetry & Remote Sensing, LAS Specification 1.4 - R14. The American Society for Photogrammetry & Remote Sensing, 2019.

[84] M. Isenburg, "Laszip: lossless compression of lidar data," Photogrammetric Engineering & Remote Sensing, vol. 79, 02 2013.

[85] ESA, "Introducing Sentinel-2," 2022.

[86] ESA, "Sen2Cor Process," 2022.

[87] COGEO Contributors, "Cloud-Optimized GeoTIFF specification," 2022.

[88] Wikipedia, "Normalized difference vegetation index," 2022.

[89] Amazon Web Services, "Registry of Open Data on AWSPowered by AWS Cloud Computing: Sentinel2," 2022.

[90] T. Goulden, "Calculate Vegetation Biomass from LiDAR Data in Python," 2022.

[91] J. C. Jenkins, "National-scale biomass estimators for united states tree species.," 2003.

10 Annex 1: Questionnaire Template

In order to evaluate the usability of one of the CloudButton core components, Lithops, the project has launched a survey, circulated among use cases, to gather their feedback about their user experience. The questionnaire was designed to be answered by those not directly involved in the project who can provide valuable feedback for improving the solution. Several aspects, from applicability to elasticity or even costs, were taken into account. Results of the survey were used to validate the proposed solution and improve the QoE.

10.1 CloudButton Questionnaire

Participating in the CloudButton project has given you the opportunity to improve your use cases testing some of the core project functionalities. Your feedback will be very valuable to evaluate and improve the toolkit you used. Please reply to the following questions and let use know your opinion. It won't take more than 20 minutes.

Select your use case:

Genomics / Metabolomics / Geospatial / Other (specify which one)

A. Applicability

Rate Lithops with regards the applicability to your use case.

Transparency in cloud computing is the capability of enabling local and remote resources to be accessed using identical operations. Rate how Lithops achieves transparency and smooth transition to remote cloud resources

Rate from 1 (Very low) to 5 (Very high)

Does Lithops achieve transparency and a smooth transition from private cloud to remote cloud resources?

Rate from 1 (Very low) to 5 (Very high)

How do you define your use case (batch analytics, interactive, streaming)? Is Lithops appropriate for this use case? Justify your answer.

Long-answer text

Did you detect any limitation when using Lithops in terms of productivity?

Long-answer text

B. Simplicity

Rate Lithops simplicity (ease of use) as a platform for developing Big Data applications.

APIs interface

Rate from 1 (Very difficult) to 5 (Very easy)

Configuration management

Rate from 1 (Very difficult) to 5 (Very easy)

Store management

Rate from 1 (Very difficult) to 5 (Very easy)

Failure management

Rate from 1 (Very difficult) to 5 (Very easy)

Resource management

Rate from 1 (Very difficult) to 5 (Very easy)

Multi-cloud backend management

Rate from 1 (Very difficult) to 5 (Very easy)

Big data pipeline management

Rate from 1 (Very difficult) to 5 (Very easy)

Integration with existing apps and libraries

Rate from 1 (Very difficult) to 5 (Very easy)

Fine grained resource management (GPUs, large VMs)

Rate from 1 (Very difficult) to 5 (Very easy)

Code development

Rate from 1 (Very difficult) to 5 (Very easy)

Data manipulation

Rate from 1 (Very difficult) to 5 (Very easy)

Productivity and time gains in resource management

Rate from 1 (Very difficult) to 5 (Very easy)

Productivity and time gains in code development

Rate from 1 (Very difficult) to 5 (Very easy)

C. Productivity

Rate Lithops productivity (efficiency achieved compared with other tools or setups that you may know) as a platform for developing Big Data applications.

APIs interface

Rate from 1 (Very low) to 5 (Very high)

Configuration management

Rate from 1 (Very low) to 5 (Very high)

Store management

Rate from 1 (Very low) to 5 (Very high)

Failure management

Rate from 1 (Very low) to 5 (Very high)

Resource management

Rate from 1 (Very low) to 5 (Very high)

Multi-cloud backend management

Rate from 1 (Very low) to 5 (Very high)

Big data pipeline management

Rate from 1 (Very low) to 5 (Very high)

Integration with existing apps and libraries

Rate from 1 (Very low) to 5 (Very high)

Fine grained resource management (GPUs, large VMs)

Rate from 1 (Very low) to 5 (Very high)

Code development

Rate from 1 (Very low) to 5 (Very high)

Data manipulation

Rate from 1 (Very low) to 5 (Very high)

Productivity and time gains in resource management

Rate from 1 (Very low) to 5 (Very high)

Productivity and time gains in code development

Rate from 1 (Very low) to 5 (Very high)

D. Scalability, Elasticity and Performance

Rate how useful the toolkit was for your experiments.

Could you scale correctly your experiments?

Rate from 1 (Completely disagree) to 5 (Totally agree)

Was the maximum parallelism reached in compute functions enough in terms of performance

Rate from 1 (Completely disagree) to 5 (Totally agree)

Were the maximum data volumes processed in your experiments enough in terms of performance?

Rate from 1 (Completely disagree) to 5 (Totally agree)

Was the reduction experimented in execution time enough in terms of performance?

Rate from 1 (Completely disagree) to 5 (Totally agree)

Was the benefit from managing more data per second enough in terms of performance?

Rate from 1 (Completely disagree) to 5 (Totally agree)

Are your experiments dynamic in the use of resources or they use always the same resources?

Rate from 1 (Completely disagree) to 5 (Totally agree)

Did you find any limitations in Lithops in terms of scalability/elasticity?

Long-answer text

E. Cost

Rate the savings you found in your experiments.

Improvements in your experiments' costs in terms of cloud resources.

Rate from 1 (Very low) to 5 (Very high)

Improvements in the development cost of your experiments.

Rate from 1 (Very low) to 5 (Very high)

Improvements in the cloud management cost of your experiments: provisioning and configuration.

Rate from 1 (Very low) to 5 (Very high)

Improvements with respect previous technologies used: cluster computing, dedicated computing, VMs.

Rate from 1 (Very low) to 5 (Very high)

Improvements from the use of pay-as-you-go serverless model.

Rate from 1 (Very low) to 5 (Very high)

Did you detect any limitation when using Lithops in terms of costs?

Long-answer text

F. Learning and documentation

Rate how easy it was to use the toolkit with the available information.

In general terms, did you find Lithops easy to use?

Rate from 1 (Very difficult) to 5 (Very easy)

In case you made use of Lithops documentation, was it easy to understand and reliable?

Rate from 1 (Very difficult) to 5 (Very easy)

In case you made use of Community support, was it helpful and reliable?

Rate from 1 (Very difficult) to 5 (Very easy)

Did you find any problem when learning or debugging Lithops? Please, explain your answer.

Long-answer text

How user friendly (in terms of usability) did you find Lithops?

Rate from 1 (Very difficult) to 5 (Very easy)

How useful do you find Lithops to scale applications?

Rate from 1 (Not useful) to 5 (Very useful)

Do you think you have been able to do the same work in the same time without using Lithops?

Yes / No

G. System Evaluation

Rate your user experience and share your suggestions for improvement.

How would you rate your experience using Lithops?

Rate from 1 (Not satisfying) to 5 (Very satisfying)

Which functionalities do you consider as the most useful ones?

Scalability / Performance / Elasticity / Parallelization / Productivity / Other (specify which one)

Do you have any suggestion for improving the system?

Long-answer text

Is there any functionality not considered within the system that you see as a 'nice-to-have' one?

Long-answer text

How do you assess the benefit from releasing resources management overhead?

Long-answer text

11 Annex 2: Answers to the CloudButton Questionnaire

CloudButton survey has been filled up by 16 participants representing the use cases, but not directly involved in the project development.

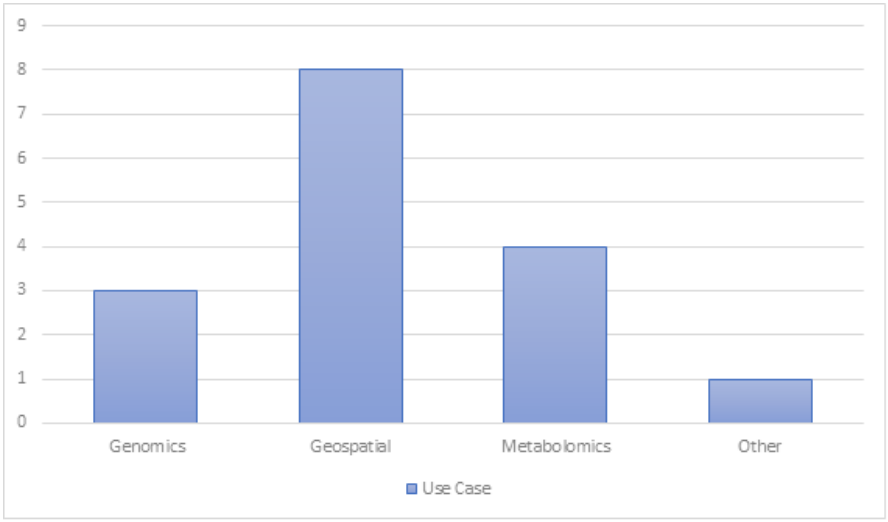


Figure 61: Participants in the survey

Surveyees used Lithops to perform tests on their use cases for the first time, without having any prior knowledge about it. In this way, they provided a 'clean' view on how the tool performs, identifying the strongest points and how it can be improved.

There is only one participant selecting Other, and the answers belongs to distributed computing in general.

Participants rated Lithops in seven different aspects, and results are as follows:

11.1 Applicability

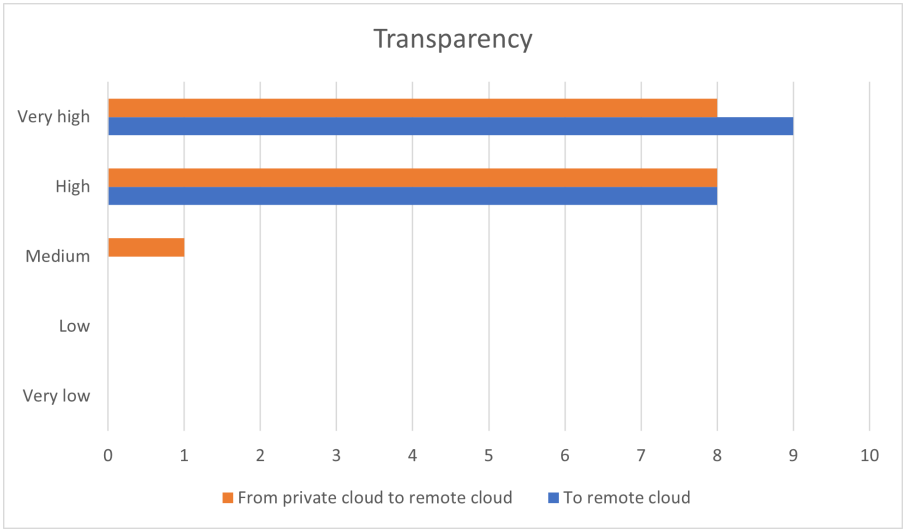


Figure 62: Lithops transparency rating

9 out of 16 participants considered Lithops has a very high level of transparency in its operations when transitioning for the first time to the cloud. While the rest considered that the level is still high. When rating the transition from private cloud resources to remote cloud resources, answers are quite

similar. This means that all participants positively rate the usage of Lithops for a smooth transition to remote cloud resources.

In order to understand how much applicable Lithops is to different scenarios, they survey asked about the different nature of the use cases.

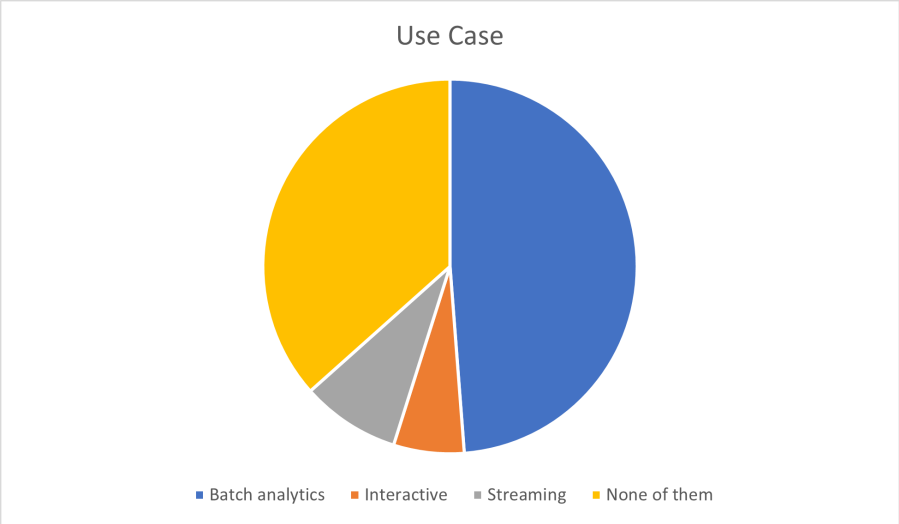


Figure 63: Types of use cases

All the respondents consider that the tool is appropriate for their use cases, highlighting the benefits parallelism and scalability functionalities are bringing to them, in terms of efforts, costs and invested time. Regarding its comparison with other tools, respondents find very satisfying the absence of need for installing other vendor specific tools, such as CLI or SDK, while they are missing the integration with other intermediate tools, such as redis. At the same time, 11 out of 16 respondents did not find any significant limitation preventing them fro using Lithops in the near future. Regarding the others, 1 considered that the quality of the documentation can be improved. As by the time Lithops was tested it was still under development, further improvements in the documentation have been provided. 2 considered that configuring it was not an easy task. Further instructions for addressing this specific issue will be provided in the final version of the documentation. And finally, the other 2 found difficult to import custom libraries when creating the Docker image.

11.2 Simplicity

Another aspect about Lithops to be assessed is its ease to use compared with other existing tools.

In terms of simplicity, or how easy was to use Lithops, most of the respondents valued positively the provided interface. At the same time, failure and multi-cloud backend management got the highest scores in terms of ease of use. Other aspects, such as storage, resource and big data pipeline management were also well valued in terms of simplicity. However, there were some critics regarding the configuration management. Although it still got high scores, some users found it a little bit complicated to use. Finally, it was quite appreciated the possibility of easily integrate it with already existing applications and libraries and the additional gains in terms of resource management and code development.

11.3 Productivity

Participants in the survey also evaluated the efficiency of Lithops compared with other tools.

As it happened while evaluating the simplicity of the tool, most of the respondents considered as one of Lithops strongest points its APIs interface. Other very well valued aspects were the productivity and time gains in resource management and code development, what increases productivity not only on the application, but on design time. However, there are some aspects like failure man-

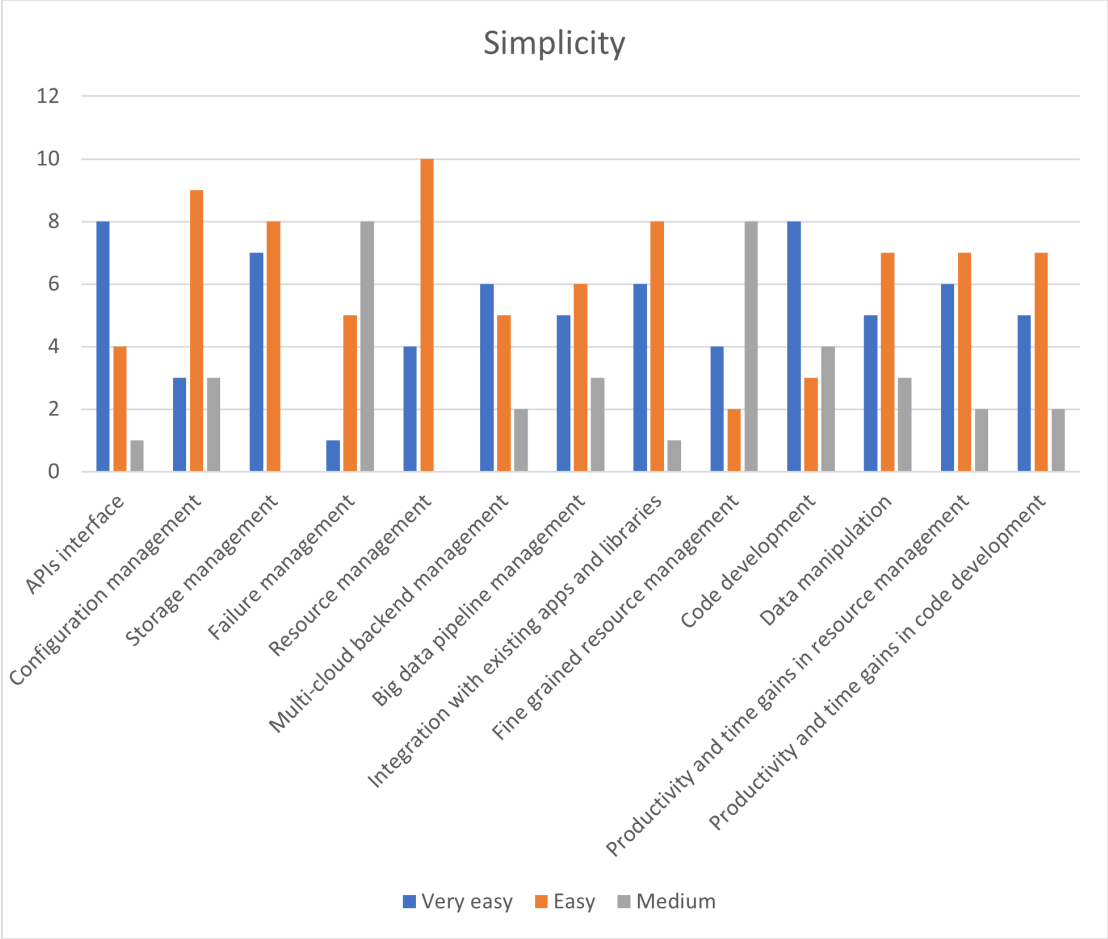


Figure 64: Ease of use

agement or fine-grained resource management that needs to be improved. Overall, the evaluation of the tool was pretty good taking into account the given maturity of the solution.

11.4 Scalability, Elasticity and Performance

Participants in the survey rated Lithops in terms of the scalability achieved in their experiments, taking into account that 14 out of 16 of them are dynamic in terms of use of resources, while only 2 of them use always the same ones.

In terms of scalability, 8 out of 16 respondents totally agree with Lithops allowing a proper scaling up of their experiments, while 7 out of 16 agree with the statement. None of the respondents faced any issue scaling up any application while performing their experiments. At the same time, 8 out of 16 respondents (on average) considered enough in terms of performance the parallelism reached, the increased number of data volumes processed, and the reduction experimented in execution time, as well as the increased benefits for managing more data per second. Some of them found limitations not related to Lithops but to specific cloud resources, or time and memory limitations of given FaaS functions.

11.5 Cost

Respondents also rated Lithops in terms of cost savings within their experiments.

Most of the respondents found a significant improvement in terms of costs using Lithops for executing their experiments or even comparing it with previously used technologies, such as dedicated computers, virtual machines or cluster computing. However, there is still one weak point related to monitoring cloud billing as it maily relies on the provider and not on the tool. Thus, this additional

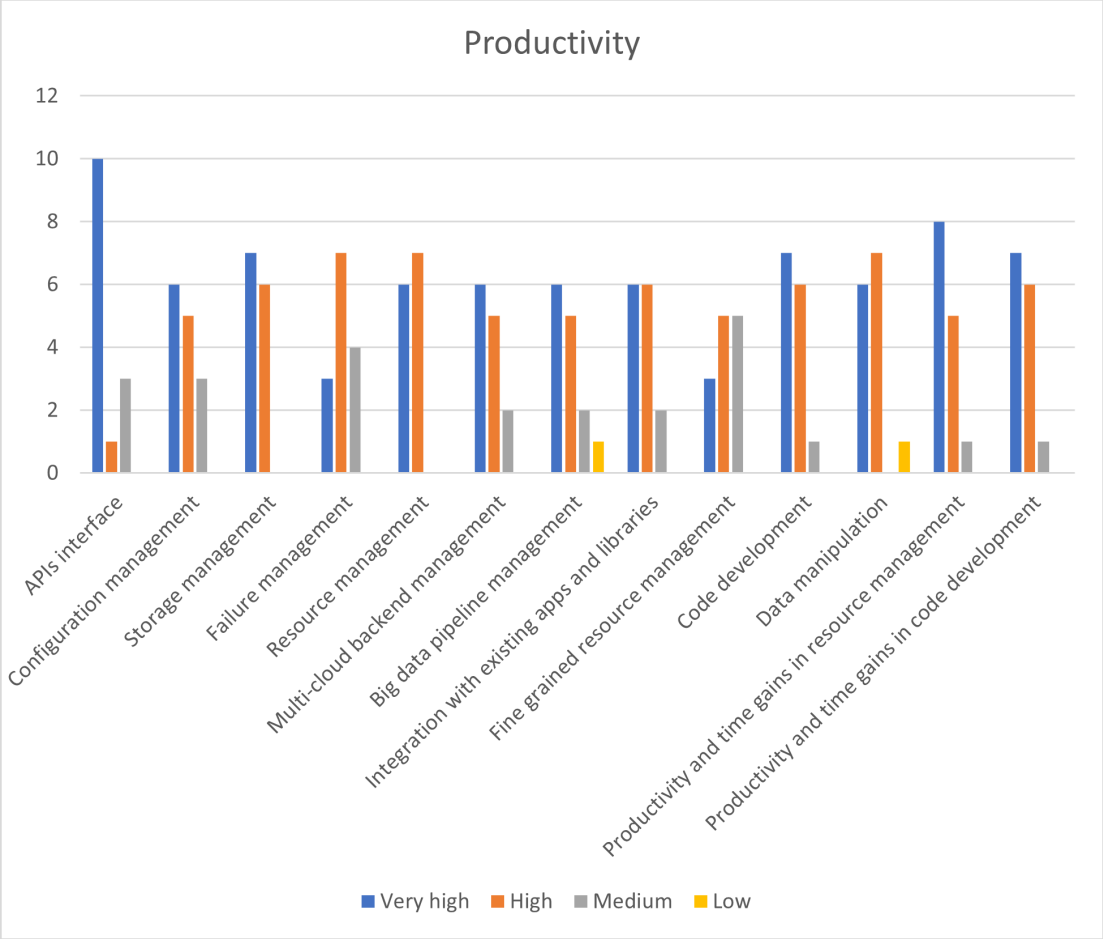


Figure 65: Lithops Productivity

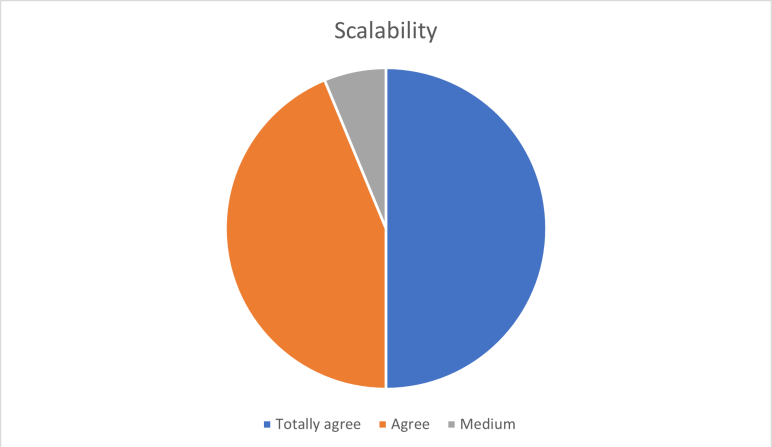


Figure 66: Proper scalability of experiments

functionality is out of the scope of the project.

11.6 Learning and documentation

Another aspect to evaluate was the available documentation and the quality of the information provided to use Lithops toolkit.

According to the participants in the survey Lithops toolkit is easy to use, user friendly and very

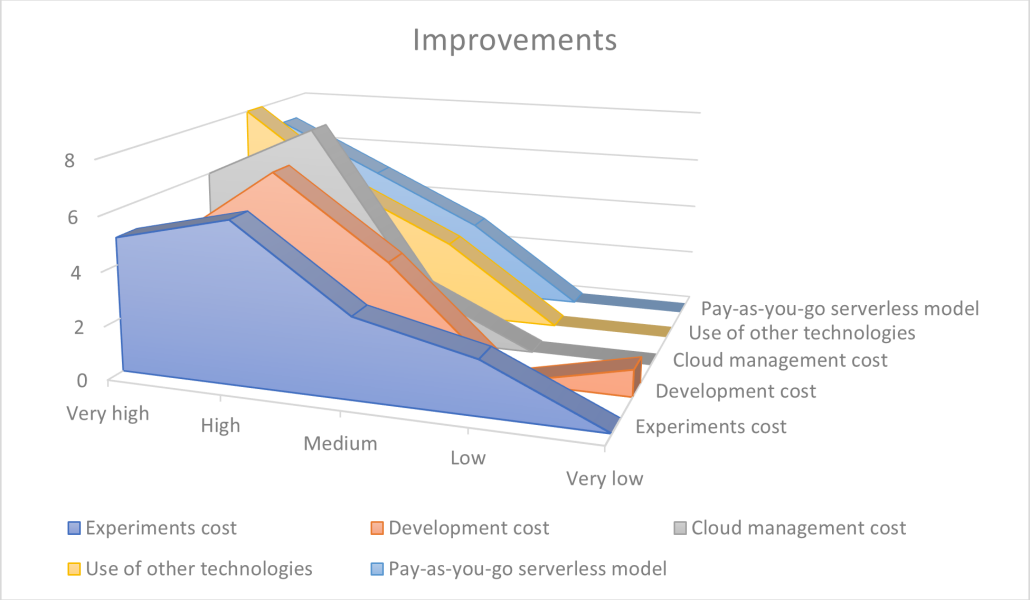


Figure 67: Improvements in terms of cost

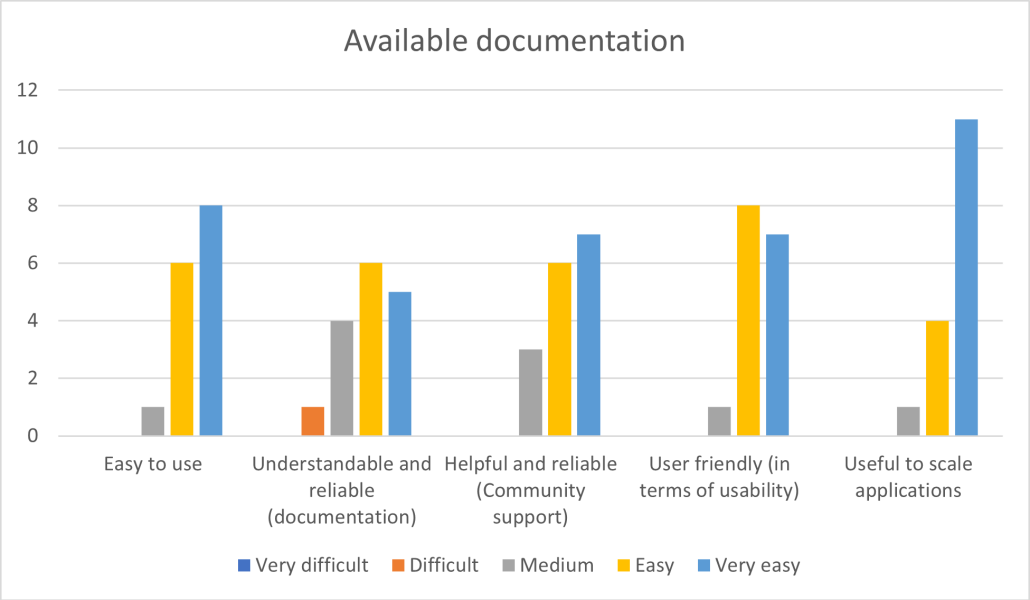


Figure 68: Lithops documentation assessment

useful to scale applications. Community support was also very well valued by those who made use of it. However, the existing documentation was sometimes difficult to follow due to a lack of examples or debugging information. Since the time performing the experiments, a new version of the documentation has been released to make it easier to follow for external readers. At this stage it is important to highlight that only 2 out of 16 respondents considered that they will be able to do the same work at the same time without using Lithops. While the rest considered that used Lithops significantly reduced the amount of work and time needed to develop the applications and perform the experiments.

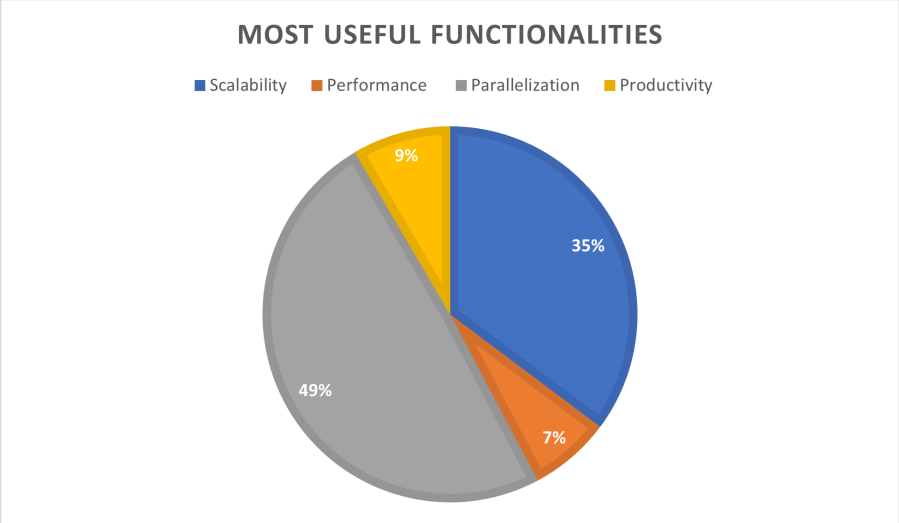


Figure 69: Overall system evaluation

11.7 System Evaluation

In general terms, 8 out of 16 respondents rated the overall experience using Lithops for performing their experiments as very satisfying, while the rest rated it as satisfying.

According to the benefits gained using Lithops to develop applications and perform experiments, nearly a half of the respondents considered parallelization as the most useful one, followed by scalability, productivity and performance. As expected, due to the evaluation of other aspects, suggestions for improvement includes better documentation and more promotion of the toolkit itself. Respondents also suggested some nice-to-have functionalities on top of the existing ones, like machine learning support or stateful processing. Overall, all of them rated the experience as very positive mainly due to time savings and increased productivity.

12 Annex 3: Serverless variant caller READMEs

12.1 Installation requirements

12.1.1 Install local dependencies (where the script is executed):

python 3.8 and `pip install Lithops\[aws\]`

redis (redis-cli)

edirect (if using SRA option):

```
sh -c \
  "$(wget -q ftp://ftp.ncbi.nlm.nih.gov/entrez/entrezdirect/install-edirect.sh -O -)"
```

Important to have the new script added binaries to the PATH variable of the shell used to launch the variant caller.

12.1.2 Build and upload the runtime

```
cd dockerfile && \
Lithops runtime build -f Dockerfile -b aws_lambda lumimar/hutton-genomics-v03:13
```

12.2 Running the variant caller

This is a command line example in which all configurable parameters are made explicit, although many have default values that can be used so that only few parameters are actually required.

```
python varcall_Lithops_demo_v5.py -fq ERR9729866 -fa hg19.fa -cl aws \
  -b cloudbutton-variant-caller-input -fb ayman-Lithops-meta-cloudbutton-hutton \
  -ds SRA -nfq 2000000 -nfa 100000000 -ofa 300 -rl 152 -t 0 -ff csv -S3w False \
  -rt lumimar/hutton-genomics-v03:18 -rtm 4096 -rtr 4096 -bs 75% -ftm 900 -ftr 900 \
  -sk False -ip 54.146.89.181 -id i-0cee52f66655d990b -rg us-east-1
```

A wrapper script, `run_variant_caller_v2.sh`, allows to run the variant caller with these additional features: - inputting command line arguments from a table `varcall_args.tsv` - sorting live function log by function number for readability - extraction of summary stats for the run, to quickly identify where the run might be failing - summary stats linked to log file and to command line arguments - generation of plots with time of various pipeline stages and size of intermediate files generated - specifying command and run id, and how many repeats of the script to run, if testing reproducibility.

Here is a command line example:

```
bash run_variant_caller_v2.sh varcall_args.tsv 1 1 True
```

The four command lines arguments are: 1. variant caller argument table, 2. run id (if same settings run multiple times), 3. number of iterations (if script run multiple times), 4. whether to run the variant caller (True) or just process the log file (False). The latter is useful in the event of an aborted run, as it processes the existing incomplete log and generates plots and info to quickly identify where a problem arose.

Together with the sorted log and a series of tables (all saved to the `varcall_out` subfolder), the script also adds lines to the summary file `varcall_results_summary.tsv`, providing basic stats about the number of functions that "made it" through the various stages of the pipeline. Its output also includes the initial commandline, and is also present at the end of the log file.

The args `varcall_args.tsv` has two header lines, with short and long argument option names (presented here in column format, first two columns in table below)

short option name	long option name	explanation	example
run_n	#NA	NA	#1
fq	fq_seq_name	FASTQ sequence name	SRR15068323
fq1	FASTQ1	FASTQ file name 1	NA
fq2	FASTQ2	FASTQ file name 2	NA
fa	FASTA	FASTA file name	hg19.fa
cl	cloud_adr	cloud provider	aws
b	bucket	bucket	cloudbutton-variant-caller-input
fb	fbucket	bucket for FASTA file	ayman-Lithops-meta-cloudbutton-hutton
ds	data_source	SRA or S3	SRA
nfq	FASTQ_read_n	number of reads per FASTQ chunk	800000
nfa	FASTA_char_n	number of characters per FASTA chunk	100000000
ofa	FASTA_char_overlap	overlap between FASTA chunks	300
rl	read_length	read length (to calculate approx FASTQ size)	152
t	tolerance	number of additional strata to include in alignment output	0
ff	file_format	`mpileup` file format conversion choice (csv or parquet)	csv
itn	iterdata_n	Number of functions to launch (all if empty)	5
S3w	temp_to_S3	saving temporary files to S3 for debugging	FALSE
rt	runtime_id	lambda function runtime id	lumimar/hutton-genomics-v03:18
rtm	runtime_mem	memory associated with map function	4096
rtr	runtime_memr	memory associated with reduce function	4096
bs	buffer_size	size of buffer in reduce function	75%
ftm	func_timeout_map	timeout for map functions	900
ftr	func_timeout_reduce	timeout for reduce functions	900
sk	skip_map	skip map function if debugging reduce phase	FALSE
ip	ec2ip	ec2 IP for redis	54.146.89.181
id	ec2id	ec2 id for redis	i-0cee52f66655d990b
rg	ec2region	ec2 region	us-east-1

12.3 Running the variant caller using Docker on AWS EC2

Here, we provide instructions to run the variant_caller client using a pre-built Docker image running in an AWS EC2 VM. Docker is a platform that allows you to build and deploy packaged software environments ("containers"). Combining this with the use of AWS Virtual Machines (VMs), you can get up-and-running quickly, without the need to leverage local resources.

12.3.1 Virtual Machine configuration

VM setup on AWS

Configuring the virtual machine instance follow the same process used for configuring the redis server, with only a small exception with respect to the storage configuration (see below). Note that the below steps have only been tested on the Ubuntu 20.04 instance type, which is free-tier eligible, thus we recommend this as the OS selection.

To set up your AWS account for running the Lithops variant caller in a VM, see this guide.

Once your account is setup, reference section 1 (Virtual Machine Configuration) from the redis-server setup documentation.

Storage configuration

When selecting your disk storage size, the total size will at minimum need to be large enough to host the docker image (tkchafin/varcall_client). Uncompressed, this is >3GB. Additionally, sufficient space will be needed to temporarily store the input FASTQ file in `/tmp` during the pre-processing phase, unless indexes are already present in the S3 bucket.

Connecting to the VM via SSH

If configured correctly to allow incoming SSH traffic (see section 1.4: Network Configuration from the redis-server documentation), you should be able to now SSH directly to the running instance using the private key file specified in the VM configuration: `ssh -i private_key.pem ubuntu@<VirtualMachine_DNS>`

Note that the username is `ubuntu` and the may be found in the instance summary, under "Public IPv4 DNS" (it will look like `ec2-##-###-##-##.compute-1.amazonaws.com`).

After running this, you should notice your terminal prompt has changed to:

```
ubuntu@ip-###-##-##-##:~$
```

12.3.2 Docker configuration in the VM

Once you have verified that you can successfully access the VM instance from your local machine, you are ready to configure the docker. Note that here there are several steps which will need to be completed either from the your local computer, and others from the VM, accessed using SSH (see Section 1.3 above).

For convenience, we recommend opening two terminals, one running locally, and another connected to the VM via SSH.

Installing docker in the VM

This step occurs on the **VM**. Here, you will install docker. Docker installation follows the Docker documentation, with commands excerpted below for convenience (with some slight modifications):

```
# ssh to the running VM instance
ssh -i private_key.pem ubuntu@<VirtualMachine_DNS>

# remove previous versions (just in case)
sudo apt-get remove docker docker-engine docker.io containerd runc

# set up the repository
sudo apt-get update && sudo apt-get install -y ca-certificates curl gnupg lsb-release

# add Docker's official GPG key
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

# set up repo
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
    https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# install docker engine
sudo apt-get update && \
  sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin

# add user to docker group
sudo usermod -a -G docker ubuntu
```

Now, you can pull the pre-built docker image from our public DockerHub repository:

```
# docker pull
docker pull tkchafin/varcall_client:0.2
```

If you want to make modifications to the docker image, or build it yourself, see Section 3: Building the container from scratch (below).

Install variant_caller code on the VM

Next, you will need a copy of the variant_caller repository on the **VM**. From your **VM** SSH terminal, first install git and then clone the repo:

```
# install git
sudo apt-get install -y git

# clone repository
git clone https://gitlab1.bioss.ac.uk/lmarcello/serverless_genomics.git

# if using a development branch, checkout that branch, e.g.,
# git checkout some_dev_branch
```

Upload user configuration files to VM

Now, from your **local** terminal, you need to upload your Lithops configuration and AWS credentials files.

Examples of both are provided in the variant_caller repository at

`./serverless_genomics/variant_caller/varcall_client.`

To upload them, you can simply use `scp`, pointing to your private key file (the same used for SSH access to the VM):

```
#upload aws credentials
scp -i private_key.pem <1: 1 windows (created Wed Jun 29 17:31:54 2022)
/path/to/.aws/credentials> ubuntu@<VirtualMachine_DNS>:/root/aws_credentials

# upload Lithops config
scp -i private_key.pem </path/to/.Lithops/config> \
  ubuntu@<VirtualMachine_DNS>:/root/Lithops_config
```

Running the Docker container

From the **VM** terminal, you can then run the Docker. We have provided a convenience script for you, which "mounts" the necessary files from the EC2 environment so that they are accessible within the container.

These are:

EC2 Path	Container Path	Description
/tmp	/tmp	This is where the Lithops logs will be written (to /tmp/Lithops/logs)
/home/ubuntu/serverless_genomics	/root/serverless_genomics	Copy of the variant_caller gitlab repository. This contains the code for running the Docker container, as well as the code for running the pipeline
/home/ubuntu/aws_credentials	/root/credentials	AWS credentials file (containing your super-secret key information). Inside the container, this is set to the environmental variable AWS_SHARED_CREDENTIALS_FILE
/home/ubuntu/Lithops_config	/root/config	Lithops configuration file, containing the details of your serverless execution. Inside the container, this is set to the environmental variable Lithops_CONFIG_FILE

You can run the Docker from the **VM** home directory simply by providing these paths as arguments for the convenience script (note you also provide the container image name, e.g., `tkchafin/varcall_clien`

```
./serverless_genomics/variant_caller/varcall_client/varcall_docker_run.sh \
  ./serverless_genomics/ ./aws_config ./litops_config tkchafin/varcall_client:0.2
```

That's it! You now should notice your terminal prompt has changed to something like:

```
root@4d311b7ae8e8:~/serverless_genomics#
```

You can now run the pipeline.

Running the Docker container in a background shell

Although you can now run the pipeline, if you logout or exit your **VM** terminal, any ongoing pipeline runs or running images will stop. To prevent this, you can run a background shell using the tool `tmux`:

```
# start a tmux instance
# should notice a green bar at bottom of terminal after running this
tmux


# run the docker
./serverless_genomics/variant_caller/varcall_client/varcall_docker_run.sh \
  ./serverless_genomics/ ./aws_config ./litops_config tkchafin/varcall_client:0.2
```

As above, your terminal prompt should change to indicate you are "in" the container. To detach from the `tmux` instance, just type `CTRL+B`, then `D`. Now, you can safely step away for a coffee and return to have mapped and variant-called data!

To revisit your running tmux sessions, you can check for running instances using `tmux ls`:

```
ubuntu@ip-172-31-82-75:~$ tmux ls
0: 1 windows (created Wed Jun 29 17:18:28 2022)
ubuntu@ip-172-31-82-75:~$
```

Now, "re-attach" using `tmux attach`:

```
tmux attach -t 0
```

12.3.3 Building the container from scratch

If you have an issue with Docker (and can't wait for help by posting over at the Gitlab Issues page), or want to add some functionality to the Docker, you can find the complete Dockerfile at `serverless_genomics/variant_caller/varcall_client/Dockerfile`. Inside, you will find directions for building the container.

To build the docker from scratch (and with docker installed on your **local** machine), you can type:

```
# e.g. docker build -t tkchafin/varcall_client:0.2 varcall_client/
docker build -t <username>/varcall_client:<tag> varcall_client/
```

Note that this can take a while, especially if it is the first time you are building it (hence no cache for docker to use).

12.4 Redis Installation

12.4.1 Virtual Machine configuration

Virtual Machine Image (OS)

Cloud providers offer a wide variety of images depending on the needs of the user. We recommend the Ubuntu Server version.

Currently (June 2022) Lithops does not support python 3.10, as AWS lambda functions do not support this version of python, so we recommend using Ubuntu Server 20.04 (which includes Python 3.8). Even so, it is possible to use Lithops with Python 3.10 which is included in Ubuntu Server 22.04, but in this case the default runtime provided by Lithops cannot be used and a specific runtime that includes Python 3.10 will have to be created using Docker.

Both operating systems are included in the AWS Free Tier.

Instance type

The choice of the type of virtual machine to use depends on the requirements of your application and your experiment. We recommend using the machines included in the AWS free tier, and if necessary, use a more powerful machine (family, cpu, memory).

To start getting familiar with these virtual machines you can use the t2.micro machine from the t2 family (general purpose), featuring 1 virtual CPU and 1 GiB of memory.

Login Key Pair

To access the virtual machine and control it from your personal computer, you need to generate a unique pair of keys that will authenticate you when connecting to the machine via the ssh protocol. This key pair can be generated using the AWS console, and you will need to carefully safeguard your private key on your computer.

In order to connect to the Virtual Machine from your personal computer, you can use the command:

```
ssh -i private_key.pem ubuntu@<VirtualMachine_IP>
```

Network configuration

Network configuration is crucial to keep your virtual machine secure and functional. Below we will detail the configuration used for the genomic use case.

We will allow all outgoing traffic and control all incoming traffic.

Outgoing traffic rules:

Type	Protocol	Port interval	Destination
All the traffic	All	All	0.0.0.0/0 (All)

Incoming traffic rules:

Type	Protocol	Port interval	Origin	Description
SSH	TCP	22	0.0.0.0/0 (All)	SSH for admin Desktop
TCP customized	TCP	6379	0.0.0.0/0 (All)	Redis (protected with password)
TCP customized	TCP	11222	0.0.0.0/0 (All)	Infinispan (alternative to redis)
TCP customized	TCP	2375	0.0.0.0/0 (All)	Docker daemon
TCP customized	TCP	2376	0.0.0.0/0 (All)	Docker daemon encrypted

Storage configuration

In the event that disk storage is needed, the necessary gigabytes of storage can be allocated thanks to a virtual disk (EBS in the case of AWS). We recommend using the minimum necessary disk, and increasing its size depending on needs.

```
1 x * GiB gp2 root volume (Example for AWS EC2)
```

Static IP

If you want your virtual machine to have a fixed/static IP, you can create a new elastic IP (in the case of AWS) and assign it to your recently created virtual machine.

12.4.2 Installation and Configuration of software used in the Virtual Machine

Clone this repository and start configuring your VM After cloning the repository move all its content to your user directory

```
mv CloudButton-Redis-Installation/* /home/user_name/
```

Execute redis_server_installation_1.sh

```
sudo ./redis_server_installation_1.sh redis_password
```

Execute redis_server_installation_2.sh

```
./redis_server_installation_2.sh redis_password
```

Fill out the configuration needed by redis_server_installation_2.sh

In the command prompt from `/home/$(logname)` you should invoke your favourite text editor, and fill out the requested fields. For instance:

```
vi .Lithops/config
```

After that, you should configure the AWS client:

```
aws configure <AWS Access Key ID> <AWS Secret Access Key> <Default region name> json
```

12.4.3 Test correct installation

Close the terminal where you performed the installation, and open a new one, so you are using the uptdated $PATH. Test Lithops with the following command, or use the snippets of the official Lithops' repository to implement your own program.

```
$ Lithops test
```