Usama Benabdelkrim Zakan

INTELLIGENT OPTIMIZATION OF DISTRIBUTED SYSTEMS: PERFORMANCE ANALYSIS WITH LITHOPS AND MACHINE LEARNING

FINAL MASTER'S PROJECT

Directed by Dr. Pedro Antonio García López

Master's Degree in Computer Security and Artificial Intelligence Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona

2024

Acknowledgements

First and foremost, I want to express my deepest gratitude to my family. To my mother, Zohra: your support and love have carried me through everything.

I also want to thank my brothers, Adel, Walid, and Youssef, my sisters-in-law, my nieces and nephews, and all my other family members who have always been there for me.

A special mention goes to Youssef, who is expecting a child soon. This wonderful news reached me while I was writing this thesis, giving me extra motivation to keep going.

Another key person in my life is Josep Oliveras i Samitier, who has been a mentor and source of wisdom since the day I met him. He truly marked a turning point in my life.

Thank you all for making it your goal to help me achieve mine.

My sincerest thanks to my advisor, Dr. Pedro Antonio García López, for his valuable advice, guidance, and trust. His challenges have pushed me to grow as a professional, and the opportunities he opened for me have been vital to my development.

I would also like to thank my friend Robert for always believing in me. His confidence in me has been a pillar of support throughout this journey.

Lastly, I would like to thank the members of *CloudLab* for their helpful feedback and collaboration, as well as the `Lithops` open-source community for providing the tools that made this project possible. Your dedication to open knowledge and innovation has greatly inspired my work.

This journey has not been easy. As a child, I dreamed of becoming an engineer, and now I've not only completed this master's degree but also had the privilege of teaching at URV and working on cutting-edge projects like Datoma and UNICO. It's been an incredible journey, and I'm excited for what comes next.

Contents

1 Introduction

1.1 Context

The landscape of cloud computing is experiencing a significant transformation with the emergence of serverless architectures, a paradigm shift that offers unprecedented scalability and cost efficiency. In this model, organizations only pay for the resources they actually use, significantly reducing operational overhead and enabling more agile computing solutions across a variety of industries. Central to this transformation is Lithops, an open-source, cloud-native framework designed to orchestrate and execute massively parallel workloads seamlessly across multiple cloud platforms. By abstracting the complexities of serverless computing, Lithops allows developers to manage and execute large-scale, distributed tasks effortlessly. This automation of resource management and scaling across diverse cloud environments exemplifies the agility and innovation that serverless computing brings to the table.

1.2 Historical Context

The evolution of serverless computing from traditional cloud paradigms such as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) represents a fundamental shift in how technology drives business processes. Initially, these models alleviated the burden of managing physical infrastructure but still required considerable oversight for server management. The introduction of AWS Lambda in 2014 marked a significant milestone, as it eliminated the need for server management, allowing developers to focus solely on creating responsive, event-driven applications. This progression underscores the industry's ongoing pursuit of maximizing efficiency and enhancing flexibility in cloud services.

1.3 Problem Statement

Despite its apparent advantages, serverless computing introduces significant complexities during both the development and operational phases of applications. One of the most pressing challenges is the difficulty of effectively debugging and monitoring highly distributed systems. Traditional tools often fall short in addressing the ephemeral and decentralized nature of serverless functions, leading to persistent errors and notable performance bottlenecks. Furthermore, frameworks like Lithops, which operate seamlessly across multiple cloud platforms, are constrained by the lack of comprehensive monitoring tools. Feedback from users at prominent research institutions such as the Barcelona Supercomputing Center (BSC) and CloudLab has highlighted the urgent need for tools that meet the specific demands of serverless computing environments.

This thesis aims to address these challenges within serverless multicloud environments by developing an advanced debugging and profiling tool specifically tailored for the Lithops framework. The ultimate goal is to enhance Lithops'

operational capabilities, making it more efficient, scalable, and user-friendly in managing distributed serverless applications.

The main objectives of this research are as follows:

- **Design and Implementation:** The first objective is to design and implement a debugging framework that integrates seamlessly with Lithops, enhancing real-time monitoring and enabling efficient troubleshooting across platforms. This will involve developing a system that can capture and process metrics in real time, providing immediate and accurate insights into the performance of serverless functions. This capability is essential for proactively identifying and resolving issues, avoiding disruptions, and optimizing overall application performance.
- **User-Friendly Interface:** Another critical objective is to create an intuitive interface that simplifies the complexities of monitoring and debugging distributed serverless functions. This interface will be designed for developers of varying expertise, ensuring the tool is both accessible and easy to use, while also providing powerful insights into the system's operation.
- **Performance Metrics and Optimization:** The tool will be designed to capture comprehensive performance metrics, including CPU usage, memory consumption, and network latency. These metrics are crucial for pinpointing and addressing performance issues, ensuring that the system operates efficiently even under variable workloads.
- **Integration with Machine Learning for Validation and Optimization:** This research also introduces a machine learning module, developed independently from Lithops, to validate the profiler's utility and explore optimal parallelization strategies. By keeping this module separate, Lithops remains lightweight and focused on its core functionalities. The machine learning model will analyze execution data to identify patterns and predict optimal resource allocation, serving as a powerful tool for optimization and predictive analysis.
- **Long-Term Enhancements and Scalability:** The research also aims to propose strategies that not only enhance immediate operational efficiency but also support the long-term sustainability and scalability of serverless applications. Implementing automatic scalability features will enable the system to adjust its resources and monitoring capabilities based on workload, which is critical for maintaining operational efficiency in serverless environments where load variations can be frequent and unpredictable.
- **Alignment with Lithops' Open-Source Philosophy:** Maintaining the development of this solution within the open-source ecosystem is a key objective. The goal is to ensure the tool remains accessible, extensible, and collaborative, aligning with Lithops' philosophy of promoting community-driven innovation. This will guarantee that the tool continues to evolve through active contributions from the global developer community.

- **Contribution to Run Lithops Cloud:** Additionally, this thesis will contribute to the development of another project, Run Lithops Cloud, a SaaS solution designed to run workloads through a web interface. The integration of a dedicated monitoring dashboard and the inclusion of the profiler within the Lithops version deployed on Run Lithops Cloud are key elements of this contribution.

By achieving these objectives, this research will significantly enhance the operational capabilities of the Lithops framework, improving the developer experience and optimizing the efficiency of serverless computing environments. The innovative integration of machine learning to determine ideal parallelization levels underscores this thesis's commitment to bridging cutting-edge data analysis with practical technological applications.

2 State of the Art

2.1 Overview of Serverless Computing and Monitoring Needs

2.1.1 Context and Relevance

Serverless computing has transformed cloud computing paradigms by allowing organizations to execute code in response to events without managing the underlying server infrastructure. This model supports automatic scaling and charges users solely for the compute time consumed, resulting in substantial cost savings and enhanced operational flexibility [6]. However, this dynamic environment introduces complexities that necessitate advanced monitoring tools, particularly in multi-cloud configurations managed by frameworks like Lithops [28].

2.1.2 Importance of the Topic

As serverless architectures gain increasing adoption across diverse industries, the need for robust, adaptable monitoring tools becomes critical. These tools must effectively handle the ephemeral and distributed nature of serverless functions across multiple cloud platforms. For Lithops, a cloud-native, multi-platform framework, the ability to monitor and debug across diverse cloud environments is crucial for ensuring operational efficiency and reliability [30].

2.2 Review of Existing Monitoring Tools and Technologies

2.2.1 Existing Tools

Several commercial monitoring tools, such as AWS X-Ray, Google Cloud Operations Suite, and New Relic, are widely employed in traditional cloud environments. AWS X-Ray offers deep insights into AWS services but lacks cross-platform functionality, rendering it unsuitable for Lithops, which requires multi-cloud monitoring capabilities [9]. Similarly, Google Cloud Operations Suite provides comprehensive monitoring within Google Cloud but is limited in its application to multi-cloud frameworks like Lithops [45]. New Relic, although powerful, presents challenges related to configuration complexity and cost, and its proprietary nature conflicts with the open-source ethos of Lithops [17].

2.2.2 Evaluation of Effectiveness

Existing monitoring tools, whether commercial or open-source, are generally inadequate for use in Lithops. These tools were originally designed for more static, single-cloud environments and therefore lack the flexibility and agility needed to monitor and manage serverless applications across multiple cloud platforms. A comprehensive evaluation by [41] revealed that current solutions do not sufficiently address the unique challenges posed by serverless computing, particularly in multi-cloud scenarios where Lithops operates. This shortfall is

significant, as serverless architectures demand monitoring solutions capable of dynamically adapting to the transient nature of these environments.

2.3 Limitations of Current Tools in Lithops Environments

2.3.1 Gaps in Literature

Most existing literature on monitoring solutions focuses on tools that are tightly integrated with specific cloud platforms, such as AWS, Google Cloud, or Azure. These tools lack the flexibility required by Lithops, which is designed to operate seamlessly across multiple cloud environments. While some multi-platform monitoring tools are available, they often entail considerable economic costs, making them less suitable for open-source frameworks like Lithops [31]. This highlights a critical gap in the literature: the need for cost-effective, open-source monitoring tools that cater to the multi-cloud, distributed nature of Lithops.

2.3.2 Technical Limitations

Current monitoring tools face significant technical limitations in the context of Lithops. Designed primarily for persistent, stateful applications, these tools struggle with the rapid scaling and ephemeral nature of serverless functions. The transient nature of these workloads complicates real-time data processing, resulting in scalability and performance challenges [27, 20]. Additionally, the high economic costs and proprietary nature of some multi-platform solutions render them incompatible with the open-source philosophy of Lithops [42]. This underscores the need for a dedicated monitoring solution aligned with Lithops' open-source ethos, providing multi-cloud support without imposing financial burdens on users.

2.3.3 Conclusion

While numerous monitoring tools are available for cloud environments, none fully meets the needs of multi-cloud serverless frameworks like Lithops. The lack of cross-platform support and the inability to accommodate the dynamic nature of serverless functions emphasize the necessity for a specialized monitoring solution tailored to Lithops' unique requirements. Developing a dedicated, open-source monitoring tool not only aligns with the principles of Lithops but also promotes accessibility and encourages community-driven innovation [7]. Addressing these limitations is essential for advancing the field of serverless computing and ensuring the continued success of frameworks like Lithops in complex, multi-cloud environments.

2.4 Need for Machine Learning in Optimization

2.4.1 Justification for Machine Learning Integration

Given the dynamic nature of serverless computing, there is a pressing need for machine learning models capable of predicting optimal resource utilization

based on real-time data. These models could significantly improve the efficiency and cost-effectiveness of serverless environments by dynamically adapting to workload changes [32].

2.4.2 Existing Solutions for Optimizing Serverless Performance

Several studies have explored the integration of machine learning to optimize serverless performance, particularly in predicting resource allocation and minimizing execution time. However, while these approaches yield promising results, they also present notable limitations.

Resource Allocation in Serverless Computing Research by [40] proposes the use of machine learning models to predict the optimal resource configuration based on historical job data. Although this approach reduces over-provisioning, it struggles with the high variability in execution times, leading to inaccurate predictions in dynamic environments.

Predictive Scaling and Autoscaling Predictive scaling, where machine learning models forecast required resources based on current and historical data, is another popular approach. For instance, [26] developed a system to predict the number of serverless instances necessary to handle incoming workloads. However, this solution often fails to adapt swiftly to real-time workload changes, resulting in inefficient scaling decisions.

Similarly, [14] uses reinforcement learning for autoscaling in serverless environments. Although this method shows improvements over traditional techniques, it encounters challenges in terms of training time and overhead, particularly in highly variable environments.

Function Parallelization and Execution Time Prediction Another area of research involves predicting execution time based on configuration and input size. [44] introduces a machine learning model to estimate execution time for serverless functions. However, the model fails to account for factors such as network latency and I/O, leading to inaccurate predictions in complex workflows.

Limitations of Current Approaches Many models, such as those proposed by [21] and [38], rely heavily on static data or predefined patterns, limiting their adaptability to dynamic environments. Moreover, current models frequently overlook critical factors like network conditions and function parallelization, which play significant roles in performance, especially in complex pipelines such as geospatial data processing.

2.4.3 Proposed Research on Machine Learning Models

This research proposes the development of a machine learning model that leverages runtime configuration data (e.g., number of files, input size, memory, CPU) to predict execution time. The model aims to determine the optimal level of

function parallelization (tiles) to minimize job duration. Unlike previous approaches, our model will be tested in real-world serverless environments, ensuring both adaptability and accuracy.

2.4.4 Expected Contribution and Innovation

The integration of this machine learning model into the proposed monitoring tool is expected to establish a new standard in the field. By enabling predictive monitoring and real-time optimization, the tool addresses the limitations identified in previous research and offers a robust solution to enhance serverless operations [42].

3 Lithops Profiler

3.1 Introduction to Lithops Profiler

3.1.1 Overview of the Problem

In cloud computing, serverless architectures have transformed the way applications are developed and deployed, allowing developers to focus on business logic while cloud providers manage the underlying infrastructure. However, this shift brings significant challenges, especially in monitoring and debugging applications due to the dynamic, short-lived, and distributed nature of serverless functions.

One of the main challenges when managing applications in multicloud environments using frameworks like Lithops is the complexity of effective monitoring and debugging. Lithops allows the execution of distributed workloads across multiple cloud platforms, which provides flexibility and scalability. However, this also adds extra challenges for monitoring, as serverless functions are short-lived and highly dynamic, making it difficult to capture performance metrics and identify issues in real-time, particularly across different cloud platforms.

3.1.2 Summary of the Limitations of Current Solutions

Current monitoring solutions, while effective in traditional environments, have several limitations when applied to dynamic serverless architectures, especially in multicloud settings. The key limitations are:

1. **Lack of Multicloud Integration:** Commercial tools such as AWS X-Ray and Google Cloud Operations Suite are deeply integrated within their respective cloud ecosystems, offering advanced monitoring capabilities. However, these tools are not suitable for a multicloud environment like Lithops, as they cannot effectively operate across multiple cloud platforms at the same time.

   This lack of multicloud compatibility forces users to rely on different tools for each cloud provider, increasing complexity and making it harder to get a centralized, clear view of the overall state of distributed applications.

2. **Real-Time Monitoring and Debugging:** In serverless architectures, where functions can scale up within seconds and run for very short periods, it is essential to have monitoring tools that can capture and analyze metrics in real-time. Prometheus has become a leading solution for high-speed metric collection, thanks to its real-time data handling and integration with systems like Grafana, which provides advanced visualization.

   Moreover, Lithops already has integration with Prometheus Pushgateway, making it easier to send basic information from serverless functions to a centralized system. This integration makes Prometheus a logical choice for continued use in this environment, taking advantage of the already established infrastructure for efficient, scalable monitoring.

3. **Alignment with Lithops' Open-Source Philosophy:** Lithops is built on an open-source philosophy that promotes transparency, collaboration, and continuous innovation. Maintaining development within the open-source ecosystem is essential to ensure that both the user and developer communities can contribute to and benefit from real-time improvements.

   Using open-source tools like Prometheus not only aligns with Lithops' principles but also ensures that the solution remains accessible, extensible, and free from the limitations of proprietary environments. This alignment encourages wider adoption and continuous integration of new features developed by the community.

3.1.3 How the Solution Addresses Current Needs and Challenges in Serverless Computing

Serverless computing is constantly evolving, driven by the demand for more agile, scalable, and efficient applications. However, this evolution has also introduced new challenges, especially in terms of monitoring and debugging applications in multicloud environments. The solution proposed in this thesis aligns well with these current needs and challenges, offering the following benefits:

Adaptation to the Dynamic Nature of Serverless: By providing real-time monitoring and enhanced debugging capabilities, the proposed solution directly addresses the need to manage ephemeral and dynamic applications, which are key characteristics of serverless architectures. This adaptability is crucial for maintaining operational efficiency and performance in highly volatile environments.

Facilitation of Multicloud Management: As more organizations adopt multicloud strategies, the ability of this solution to seamlessly integrate across different cloud platforms is essential. This reduces operational complexity for users and improves efficiency by enabling centralized management of distributed applications.

Scalability for Agile Growth: Automatic scalability is a fundamental requirement in serverless environments, where workloads can quickly grow or shrink. The proposed solution ensures that the system can scale efficiently, maintaining performance while minimizing costs, which is especially important in mission-critical applications.

3.1.4 Conclusion

The innovative solution proposed effectively addresses current limitations and challenges in serverless computing, providing a solid foundation for future improvements. By combining real-time monitoring, multicloud integration, automatic scalability, and a strong commitment to open-source principles, this

solution is designed to meet the needs of modern serverless environments like Lithops.

Although existing monitoring and debugging tools are useful in certain contexts, they are not fully effective in dynamic and distributed environments like Lithops. The solution presented in this thesis overcomes these limitations by offering a comprehensive platform that ensures flexibility, scalability, and adaptability in multicloud environments, all while adhering to Lithops' open-source philosophy.

In summary, this solution provides developers and administrators with the necessary tools to manage serverless applications efficiently, ensuring optimal performance and preparing for the evolving demands and continuous advancements in cloud computing.

3.2 Conceptual Framework and Design Principles

3.2.1 Theoretical Foundations

Theoretical Foundations Underpinning the Proposed Solution The development of a monitoring and debugging tool for serverless multicloud environments, such as those managed by Lithops, is based on several well-established theories and concepts from cloud computing and distributed systems. These concepts are essential to effectively manage serverless architectures, distributed systems, and predictive models through machine learning. Below, the key theoretical foundations are outlined and linked to the proposed solution.

Serverless Architectures and Their Management: Definition and Characteristics: Serverless architectures allow code to be executed without requiring explicit management of the underlying infrastructure. They are characterized by automatic scaling, pay-per-use billing, and dynamic resource allocation. These features enable developers to build applications with greater flexibility, but also create unique challenges for monitoring due to the short-lived and distributed nature of the components.

Monitoring Challenges: Serverless systems are designed to scale up or down in real-time based on demand, which makes traditional monitoring techniques, relying on static infrastructure, difficult to apply. The proposed solution addresses this by providing a dynamic monitoring system that can adapt quickly to the changes in serverless functions and configurations.

Distributed Systems and Multicloud: Distributed Systems Theory: Distributed systems consist of components located across various networks that communicate and coordinate actions by exchanging messages. In a multicloud environment, these systems manage components distributed across different cloud providers, ensuring consistency and integrity.

Multicloud Interoperability: Managing a multicloud setup requires the ability to integrate services from different cloud platforms seamlessly. This

includes adopting standardized protocols to enable smooth integration of distributed applications. The proposed monitoring solution takes these principles into account, ensuring effective monitoring and debugging across multiple cloud platforms.

Real-Time Monitoring and Debugging: The Importance of Real-Time Monitoring: In serverless environments, where functions are transient, real-time monitoring is crucial. It enables the immediate detection of performance issues, quick adaptation to changing workloads, and continuous system optimization.

Prometheus and Pushgateway Integration: Prometheus is widely known for its efficiency in collecting metrics, and when combined with Pushgateway, it enables real-time data collection from short-lived components. This allows Lithops to capture metrics from serverless functions even if they exist only for a brief moment, ensuring that no critical data is lost.

Predictive Models and Machine Learning: Machine Learning in Resource Optimization: Machine learning plays a key role in predicting and optimizing resource use in complex systems. By analyzing historical data, machine learning models can predict future trends, which is especially useful for dynamically allocating resources in serverless environments.

Application in Monitoring and Scalability: Incorporating machine learning into the monitoring process allows for predicting resource needs and optimizing the execution of serverless functions. This fits with the theory of automatic scalability, where the system adjusts its resources based on model predictions, improving overall efficiency.

References to Relevant Studies and Theories Supporting the Approach Baldini, I., et al. (2017). Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing*, pp. 1-20, Springer. This study discusses serverless computing and the challenges related to monitoring and resource management in these systems.

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Pearson Education. This book provides essential insights into the principles of distributed systems, which are crucial for understanding how to manage multicloud systems effectively.

McGrath, G., & Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. *IEEE International Conference on Cloud Engineering*, pp. 1-10. This paper outlines the technical aspects of serverless architectures, emphasizing the need for innovative monitoring tools capable of managing the ephemeral nature of serverless functions.

Morris, B., & Anderson, M. (2019). Multicloud Architecture: The Future of Cloud Computing. *Journal of Cloud Computing*, 8(1), 22-35. This article provides context on multicloud management and the importance of interoperability, which underpins the design of the proposed solution.

Baset, S. A. (2012). Cloud SLAs: Present and Future. *ACM SIGOPS Operating Systems Review*, 46(2), 57-66. This paper highlights the importance of real-time monitoring and Service Level Agreement (SLA) management in cloud environments, which are key concepts in the proposed solution.

Volz, M., & Birkner, C. (2020). *Monitoring Cloud-Native Applications: Prometheus and Beyond*. O'Reilly Media. This book discusses the advantages of using Prometheus and Pushgateway for monitoring cloud-native applications, supporting the decision to integrate these tools into the proposed solution.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press. This textbook offers a solid theoretical foundation for machine learning, particularly in resource prediction and optimization, relevant to serverless monitoring.

Dean, J., et al. (2018). A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, and Beyond. *IEEE Micro*, 38(2), 21-29. This paper explores the use of domain-specific hardware and software for system optimization, which aligns with the use of machine learning to enhance serverless architecture performance.

3.2.2 Design Principles

Description of the Design Principles Guiding the Solution's Implementation The design of the profiler for serverless environments managed by Lithops is based on key principles aimed at ensuring scalability, efficiency, flexibility, and adaptability. These principles are critical to its performance across various execution environments, including both open-source settings and a dedicated SaaS platform. Below are the main design decisions.

Dynamic and Efficient Scalability

- **Principle:** The monitoring tool must scale alongside the Lithops execution environment, ensuring accurate data collection without affecting system performance.
- **Design Decision:** A profiler is deployed for each Lithops worker created, enabling detailed data collection at the worker level. This ensures the system can dynamically scale while providing a comprehensive view of resource usage and execution status.
- **Justification:** Lithops creates multiple workers across various cloud platforms, so the profiler must scale correspondingly to deliver effective monitoring without adding significant overhead.

Isolation of the Monitoring Process

- **Principle:** The monitoring process must be completely separated from the user's code execution to prevent interference.

- **Design Decision:** The profiler runs in a dedicated process, receiving the PID of the JobRunner process (responsible for executing user code) to monitor resource consumption without disrupting the main execution.
- **Justification:** This isolation ensures the primary process's performance remains unaffected by monitoring tasks, which is essential for maintaining the integrity and efficiency of applications in Lithops.

Activation Based on Log Level

- **Principle:** Users should control when to activate the profiler to avoid unnecessary overhead if detailed monitoring is not required.
- **Design Decision:** The profiler is only activated if the log level is set to debug.
- **Justification:** Allowing users to activate the profiler as needed ensures flexibility and conserves system resources during routine operations.

Profiler Frequency and Overhead Control

- **Principle:** The profiler should enable users to balance data collection frequency with system overhead.
- **Design Decision:** The profiler includes an adjustable timeout, allowing users to customize the frequency of metric collection.
- **Justification:** This control allows users to optimize monitoring based on the needs of their applications and environments, ensuring system performance is maintained.

Managing Overload in Local Environments

- **Principle:** In environments where the Prometheus server is local and scalability is limited, it is essential to minimize system overload.
- **Design Decision:** A random delay between 0 and 3 seconds is introduced to the profiler's timeout, spreading metric submission to Prometheus over time and preventing spikes.
- **Justification:** This technique reduces simultaneous load peaks, improving system stability and performance during high demand.

Optimizing Metric Submission

- **Principle:** Efficient metric submission is key to reducing system overhead, particularly in scalable environments like SaaS.

- **Design Decision:** The metric submission API was reimplemented to batch metrics instead of sending them individually, significantly reducing overhead.
- **Justification:** Batching metrics reduces the number of requests and communication latency with Prometheus, improving operational efficiency.

Accurate CPU Monitoring

- **Principle:** Precise CPU usage measurements are essential for optimizing function execution.
- **Design Decision:** A method is implemented to measure CPU usage percentages over short intervals, capturing rapid fluctuations in utilization.
- **Justification:** Accurate CPU tracking in serverless environments, where processes are short-lived, ensures performance variability is detected early, enabling timely resource adjustments.

Open-Source Ecosystem Compatibility

- **Principle:** Ensuring compatibility with open-source technologies is essential for promoting interoperability and community collaboration.
- **Design Decision:** The solution uses open-source tools like Prometheus and Pushgateway, which are already integrated with Lithops.
- **Justification:** Using open-source technologies fosters continuous evolution and adaptability, benefiting a wide range of users and developers.

Security and Access Control in Run Lithops Cloud

- **Principle:** Ensuring the security and privacy of metrics in Run Lithops Cloud is crucial, allowing each user to access only their own data.
- **Design Decision:** AWS Managed Prometheus is used with a robust access control system to ensure that users can only view metrics related to their own executions.
- **Justification:** In a shared SaaS environment, strong access control protects user data, maintaining confidentiality and data integrity.

Extensibility and Adaptability

- **Principle:** The solution must be flexible enough to adapt to future technological changes or requirements without requiring a complete overhaul.
- **Design Decision:** The profiler is designed to be modular, allowing new metrics or analysis methods to be easily added in the future.

- **Justification:** This modularity ensures the system remains relevant over time, adapting to new technologies without requiring significant redevelopment efforts.

Interoperability with Third-Party Systems

- **Principle:** The solution should integrate seamlessly with other tools and systems users may already be using.
- **Design Decision:** Standardized APIs and common data formats (e.g., JSON or Protobuf) were implemented to ensure compatibility with third-party systems.
- **Justification:** This compatibility allows users to combine the solution with their existing workflows, enhancing its utility and adoption.

Support for Retrospective Analysis

- **Principle:** It is important to support not only real-time monitoring but also retrospective analysis of collected metrics.
- **Design Decision:** Long-term storage capabilities for metrics were implemented to allow historical analysis and trend identification.
- **Justification:** Access to historical data enables in-depth analysis, helping to uncover trends that may not be evident in real-time monitoring.

Error Handling and Automatic Recovery

- **Principle:** The system must be resilient to failures and able to recover automatically without manual intervention.
- **Design Decision:** Error detection and recovery mechanisms were implemented, including retry patterns with exponential backoff, ensuring reliable metric transmission.
- **Justification:** These mechanisms ensure the system can handle temporary overloads, minimizing downtime and optimizing performance under adverse conditions.

Documentation and Ease of Use

- **Principle:** The solution must be accessible and easy to understand for users, with clear documentation and examples.
- **Design Decision:** Comprehensive documentation, including step-by-step guides and best practices, was provided to reduce the learning curve.
- **Justification:** Good documentation enhances user experience and reduces the likelihood of configuration errors, facilitating quicker adoption.

Iterative and Agile Development Approach

- **Principle:** The development process must remain flexible to accommodate new requirements or challenges as they arise.
- **Design Decision:** An iterative and agile development approach was adopted, with frequent reviews and adaptations based on feedback.
- **Justification:** This approach ensures the system evolves continuously, adapting to real user needs and preventing long-term development bottlenecks.

Prototypes for Initial Validation

- **Principle:** Key design decisions must be validated before full implementation to reduce risks.
- **Design Decision:** Functional prototypes were developed before proceeding with full-scale implementation to test and validate core concepts.
- **Justification:** Early validation through prototypes minimizes the risk of costly mistakes in later development stages, ensuring the final solution meets expectations.

Data-Driven Research for Decision Making

- **Principle:** Optimization decisions should be based on empirical data rather than assumptions.
- **Design Decision:** A data-driven approach was adopted, collecting real performance data to inform system adjustments and optimizations.
- **Justification:** This approach ensures optimizations are based on actual data, improving system efficiency and reliability.

Compatibility with Multiple Execution Environments

- **Principle:** The profiler must work efficiently across all execution environments supported by Lithops.
- **Design Decision:** A cloud-agnostic approach was adopted, allowing the profiler to function without significant changes in different environments, including local servers and public cloud providers.
- **Justification:** The ability to run the Profiler on any backend supported by Lithops is crucial for maintaining the system's flexibility and scalability in a multicloud environment. This ensures that users can monitor and optimize their serverless functions regardless of the deployment environment, as illustrated in Figure 1.

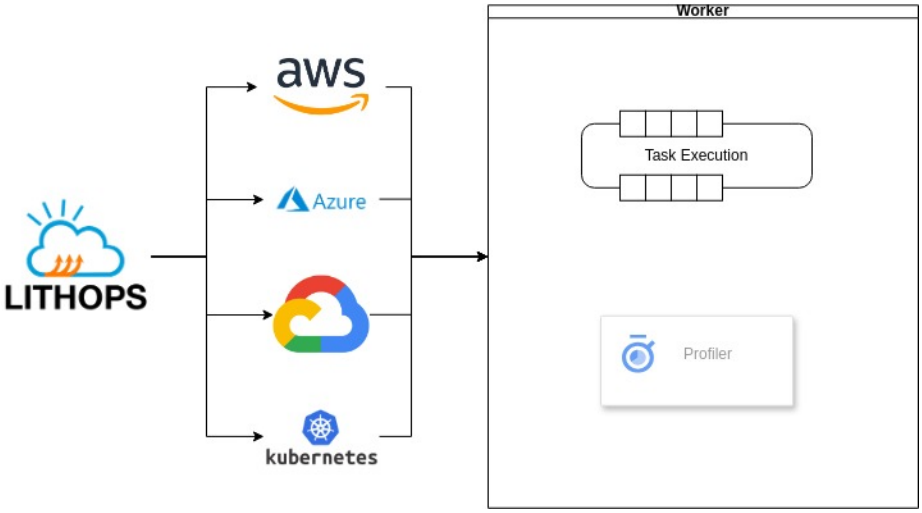

Figure 1: Diagram illustrating the Profiler's compatibility with multiple backend environments in Lithops.

Modular Approach

- **Principle:** The system should be designed modularly to facilitate long-term flexibility and maintenance.
- **Design Decision:** A modular design approach was implemented, with each component (e.g., the profiler, Prometheus integration, visualization tools) developed as an independent module.
- **Justification:** This approach simplifies updates and maintenance, as each module can evolve independently, ensuring the system remains flexible and adaptable.

User-Centered Visualization Experience

- **Principle:** Visualization tools must be intuitive and meet the needs of end users.
- **Design Decision:** A user-centered design approach was adopted, incorporating feedback from end-users to continuously improve the visualization interfaces in Grafana and Apache ECharts.
- **Justification:** Involving end-users in the design process ensures that the visualization tools are aligned with their needs, improving user satisfaction and adoption.

Notification Mechanism to Stop the Profiler Process

- **Principle:** The profiler process must automatically stop once the user's code execution is complete, avoiding unnecessary overhead.
- **Design Decision:** A notification mechanism using a pipe connection (‘parent_conn, child_conn = Pipe()‘) was implemented, allowing the JobRunner process to notify the profiler when it finishes execution, signaling the profiler to stop.
- **Justification:** This mechanism ensures that the profiler is only active when necessary, reducing system overhead and optimizing resource usage.

3.3 System Architecture

3.3.1 Overview of the Profiler Architecture

The monitoring and optimization system designed for Lithops is based on a modular architecture that enables the collection, analysis, and visualization of performance metrics in serverless environments. This architecture is designed to be efficient, scalable, and adaptable.

Main Components

- **Handler:** Responsible for creating the `JobRunner` process and collecting basic metrics during the execution of each task.
- **JobRunner:** Executes the user's code and uploads the results to the `object storage`.
- **Profiler:** The core of the monitoring system, consisting of subclasses that collect detailed metrics on CPU, memory, disk, and network usage during the execution of serverless functions.
- **Prometheus:** A metrics storage and query system that manages the data collected by the `Profiler` and `Handler`, allowing for analysis and visualization.

Data Flow and Communication The `Handler` launches the `JobRunner`, which executes the user's code while the `Profiler` collects metrics at regular intervals. After the execution is completed, the `JobRunner` uploads the results to object storage and notifies the `Profiler` to stop collecting metrics. The collected metrics are sent in real time to `Prometheus`, where they are stored for further analysis and visualization (see Figure 2).

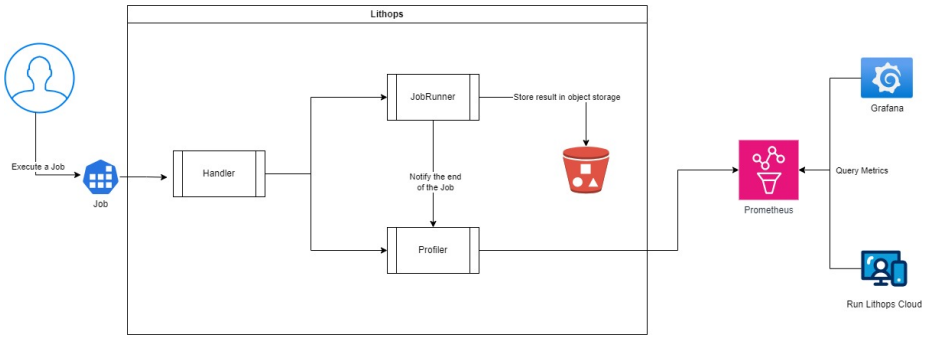

Figure 2: Diagram showing the data flow and communication between the system components.

Scalability and Security Each `JobRunner` and `Profiler` operates independently, enabling efficient scalability in environments with multiple serverless functions. As shown in Figure 3, each Lithops `worker` has its own `JobRunner` responsible for executing the user's code. At the same time, the `worker` creates another process called `Profiler`, which monitors the main execution process. This design ensures that the scalability of the `Profiler` is directly proportional to the scalability of Lithops. As Lithops scales to handle more serverless functions, more `Profiler` instances are automatically created, allowing detailed and precise monitoring of each execution.

Regarding security, the Lithops SaaS solution offers security and privacy mechanisms to ensure that all interactions with Prometheus are authenticated. Additionally, it guarantees that users cannot access metrics from other users. These mechanisms are not necessary in the native Lithops version, as the Prometheus server is autonomously controlled by each user.

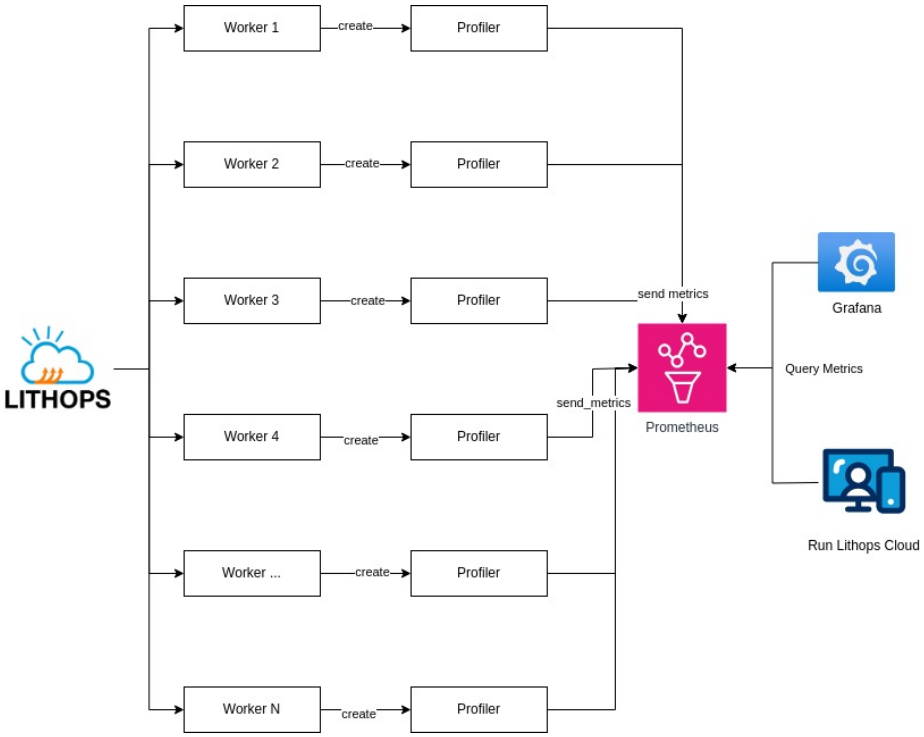

Figure 3: Scalability diagram showing how each `worker` in Lithops creates an independent `JobRunner` and `Profiler`. This allows the scalability of the `Profiler` to be directly proportional to Lithops' scalability, with each `Profiler` monitoring the process executed by its corresponding `JobRunner`.

Optimization and Resilience Optimization mechanisms such as dynamic adjustment of the metric collection interval and exponential backoff retries are implemented to ensure system resilience and minimize overhead.

3.3.2 Detailed Profiler Components

In this section, the key components that make up the Lithops Profiler are described in detail. Each of these components plays a fundamental role in the collection, processing, and storage of performance metrics, providing a comprehensive view of the behavior of serverless functions within the system. To facilitate the understanding of how these components interact with each other, a diagram illustrating the data flow and communication between them is provided below (see Figure 4).

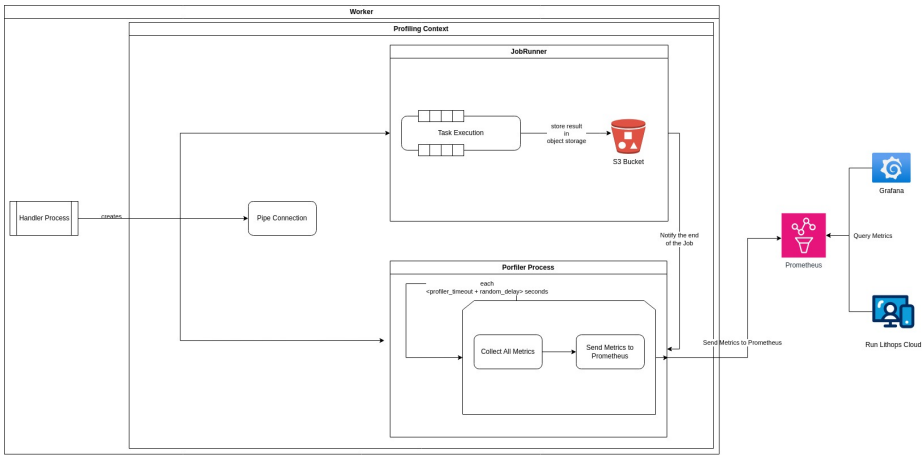

Figure 4: Detailed diagram of data flow and communication between the system components.

Handler

The `Handler` is a crucial component of the system. As shown in the diagram (see Figure 4), it is responsible for creating the `JobRunner` and `Profiler` processes using the `profiling_context` function. It is important to understand that each Lithops `worker` has its own `Handler`. Therefore, each `worker` will have a `JobRunner` process that executes the user's code and a dedicated `Profiler`. Additionally, the `Handler` collects basic metrics during the execution of each task.

Collected Metrics: The Handler gathers a set of key metrics during the execution of each job. These metrics include:

- **Timestamps:**
  - **worker_start_tstamp:** Marks the exact moment when the worker starts execution.
  - **worker_end_tstamp:** Marks the moment when the worker completes execution.
- **CPU Usage:**
  - **worker_func_cpu_usage:** Represents the average CPU usage during function execution.
  - **worker_func_cpu_system_time:** Total CPU time in system mode.
  - **worker_func_cpu_user_time:** Total CPU time in user mode.
  - **worker_func_cpu_total_time:** Sum of system and user time, representing the total CPU time used.

- **Network I/O:**
  - **worker_func_sent_net_io:** Total amount of data sent over the network.
  - **worker_func_recv_net_io:** Total amount of data received over the network.
- **Memory Usage:**
  - **worker_func_rss:** Physical memory used (Resident Set Size).
  - **worker_func_vms:** Total virtual memory used (Virtual Memory Size).
  - **worker_func_uss:** Unique memory used by the process (Unique Set Size).

Justification: These metrics are collected during any type of Lithops execution, regardless of the log level, as the collection impact is minimal. This allows users to obtain a general overview of performance without affecting code execution. For more details on the metrics collected by Lithops during any execution, you can refer to the official Lithops documentation. It is important to note that this was not part of this thesis development, as these metrics were previously collected.

JobRunner

The JobRunner is the process responsible for executing the user's code in Lithops.

Collected Metrics: During execution, the JobRunner captures several important metrics, including:

- **Function Execution Timestamps:**
  - **worker_func_start_tstamp:** Marks the start of function execution.
  - **worker_func_end_tstamp:** Marks the end of function execution.
- **Function Result Size:**
  - **func_result_size:** Indicates the size of the result produced by the function.
- **Critical Times:**
  - **worker_func_exec_time:** Total function execution time.
  - **worker_result_upload_time:** Time spent uploading function results to object storage.
- **Memory Usage:**

- **worker_peak_memory_start:** Peak memory used at the start of function execution.
- **worker_peak_memory_end:** Peak memory used at the end of function execution.

Additional Responsibility: The JobRunner is also responsible for uploading the job result to object storage after function execution. Once this task is completed, the JobRunner sends a notification to the profiler process through a pipe connection to stop collecting metrics. These metrics are also available for any type of execution.

Profiler

The Profiler in Lithops is a fundamental component for detailed monitoring of the performance of serverless functions. Below, each of the main classes and their key functions is explained in detail:

1. CPUMetricCollector Description: CPUMetricCollector is a subclass of IMetricCollector specifically designed to collect CPU usage metrics. This class is responsible for measuring the percentage of CPU usage, as well as the user and system time for a specific process.

Key Functions:

- **_collect(self, pid, parent_pid, timestamp, collection_id):** This function is responsible for:
  - Obtaining a reference to the specific process using the `pid`.
  - Measuring CPU usage over a short interval of 0.01s to avoid excessive blocking of execution.
  - Collecting other CPU times such as `user_time`, `system_time`, `children_user_time`, `children_system_time`, and `iowait_time`, which are critical for understanding the process's CPU consumption behavior.
  - Returning a `CPUMetric` instance, encapsulating all relevant CPU metrics.

2. MemoryMetricCollector Description: MemoryMetricCollector is another subclass of IMetricCollector, responsible for collecting memory usage metrics for a specific process.

Key Functions:

- **_collect(self, pid, parent_pid, timestamp, collection_id):** This function:
  - Obtains the physical memory usage (RSS) of the process in megabytes.

– Returns a `MemoryMetric` instance, encapsulating memory usage at the specified time.

3. DiskMetricCollector Description: DiskMetricCollector is responsible for collecting metrics related to disk usage, such as the amount of data read and written by a specific process.

Key Functions:

- **_collect(self, pid, parent_pid, timestamp, collection_id):** This function:
  - Collects disk operation counters.
  - Calculates the metrics `disk_read_mb` and `disk_write_mb` (in MB) and the read/write rates (`disk_read_rate`, `disk_write_rate`).
  - Returns a `DiskMetric` instance containing the collected metrics.

4. NetworkMetricCollector Description: NetworkMetricCollector specializes in collecting network metrics, such as the amount of data sent and received by a process.

Key Functions:

- **_collect(self, pid, timestamp, collection_id):** This function:
  - Obtains global network operation counters.
  - Calculates MB sent (`net_write_mb`) and received (`net_read_mb`), as well as the respective rates.
  - Returns a `NetworkMetric` instance with these metrics.

5. MetricCollector Description: `MetricCollector` is the main class that coordinates the collection of all system metrics. It acts as a container that stores and organizes the metrics collected by the `CPUMetricCollector`, `MemoryMetricCollector`, `DiskMetricCollector`, and `NetworkMetricCollector` subclasses.

Key Attributes:

- `cpu_collector`, `memory_collector`, `disk_collector`, `network_collector`: Instances of the subclasses responsible for collecting specific metrics.
- `cpu_metrics`, `memory_metrics`, `disk_metrics`, `network_metrics`: Lists that store the collected metrics.

Key Functions:

- **collect_all_metrics(parent_pid, index):**
  - **Detailed Process:**

    - Obtains a list of all child processes of the main process (`parent_pid`).
    - For each process, calls the corresponding subclasses to collect CPU, memory, disk, and network metrics.
    - The collected metrics are stored in their respective lists (`cpu_metrics`, `memory_metrics`, etc.).

- **Importance:** This function is the core of the metric collection system, ensuring that every aspect of process performance is monitored and recorded for further analysis.

- **update(self, received_data):**

  - **Functionality:** Allows the `MetricCollector` to update with new data received, ensuring that all metrics are synchronized and complete.

6. Profiler Description: The `Profiler` is the central class that coordinates the entire monitoring process. It is responsible for starting and stopping the collection of metrics, managing the interaction with the rest of the system, and sending the collected metrics to Prometheus.

Key Attributes:

- `worker_id`, `worker_start_tstamp`, `worker_end_tstamp`: Store information about the worker being monitored.

- `metrics`: An instance of `MetricCollector` that stores all the collected metrics.

- `function_timers`: A list of `FunctionTimer` that stores execution times of functions or processes.

Key Functions:

- **start_profiling(self, conn, monitored_process_pid, prometheus, job, profiler_timeout=5):**

  - **Detailed Process:**
    - Starts monitoring by configuring the process to be monitored (`monitored_process_pid`) and sets the collection interval (`profiler_timeout`).
    - Enters a loop where:
      - Collects all relevant metrics using `collect_all_metrics`.
      - Sends the metrics to Prometheus using `send_metric_to_prometheus`.
      - Checks if the monitored process has finished execution and stops profiling if necessary.

· Introduces a random delay to avoid synchronized overloads during metric collection.

- **Importance:** This function is essential for the continuous operation of the `Profiler`, ensuring that metrics are collected efficiently without interfering with the monitored process.

- **send_metric_to_prometheus(self, prometheus, key, value, type, labels_dict):**
  - **Functionality:** Formats and sends the collected metrics to Prometheus using its API.
  - **Importance:** Ensures that the collected metrics are stored and available for real-time analysis, which is critical for monitoring serverless applications in production.

- **update(self, received_data):**
  - **Functionality:** Updates the `Profiler` metrics with data received from other processes, ensuring the information is complete and up-to-date.
  - **Usage:** This is essential for maintaining data consistency when multiple instances of monitoring are performed simultaneously.

Prometheus

Prometheus is a key component of Lithops' monitoring infrastructure, acting as the main system for metric storage and queries. Its primary function is to collect, store, and process metrics sent by both the Profiler and Handler. These metrics range from CPU and memory usage to network traffic and disk operations, providing a comprehensive view of the performance and status of the serverless processes running in Lithops.

API and Configuration: The Lithops metrics API has been redesigned to integrate Prometheus directly, instead of using Prometheus Pushgateway. This allows a unified API that works both in the native version of Lithops and in the SaaS platform, Run Lithops Cloud. This change is crucial to ensure the scalability of Prometheus in production environments where multiple users may be running workloads simultaneously. In its original form, Prometheus does not effectively scale to manage large volumes of data in multi-user environments.

- **Open-Source Version of Lithops:** In this environment, users can self-manage Prometheus, installing and configuring it on local or remote servers according to their needs. This flexibility allows high customization, where users have full control over scraping configuration, storage, and alert management, ensuring the system meets the specific requirements of their operational environment.

- **SaaS Version (Run Lithops Cloud):** In the Run Lithops Cloud platform, AWS Managed Prometheus is used, a managed service that facilitates metric collection, storage, and analysis in a highly scalable and secure environment. This option efficiently manages metrics in a multi-user environment without manual intervention, thanks to the automatic scalability it offers. Additionally, to ensure data privacy and security, AWS Managed Prometheus uses IAM roles and Bearer tokens, ensuring that each user only accesses their own metrics without risk of unauthorized access.

Compatibility and Configuration: At the user level, whether opting for a self-managed Prometheus server or using managed services such as AWS Managed Prometheus, the Lithops Profiler will function optimally and without issues. The new unified API ensures the system is flexible and scalable, adapting to the needs of both local environments and large SaaS platforms.

To facilitate the configuration of Prometheus with the Lithops Profiler, a detailed guide is included in the appendix of this document. This documentation provides step-by-step instructions for configuring Prometheus in different environments.

Run Lithops Cloud Architecture: The architecture used in the SaaS version of Lithops, illustrated in Figure 5, is designed to take full advantage of AWS managed services. Below are the key components of this architecture:

- **Web Application and Lithops Client/Worker:** The interaction starts with a web application or a Lithops client/worker that submits a request for execution or monitoring.
- **Metrics API:** The metrics API receives these requests and sends them to a `Metrics Queue` to be processed asynchronously. The API also validates and signs query requests before sending them to Managed Prometheus.
- **Metrics Queue and Validation:** Metrics are temporarily stored in a queue, where they are processed and sent to Managed Prometheus using the `RemoteWrite` protocol.
- **New Executor Handler and JWT Authorizer:** A new executor handler registers each executor in an `Executors Table`, and the JWT Authorizer ensures that each request is authenticated using JWT signing, with keys managed via AWS KMS.
- **Managed Prometheus:** Finally, all collected and validated metrics are stored in AWS Managed Prometheus, where they are available for instant or range queries.
- **Security and Scalability:** As mentioned, the entire infrastructure is secured using AWS Identity and Access Management (IAM) and

automatically scaled to handle increases in demand, maintaining the availability and performance of the system.

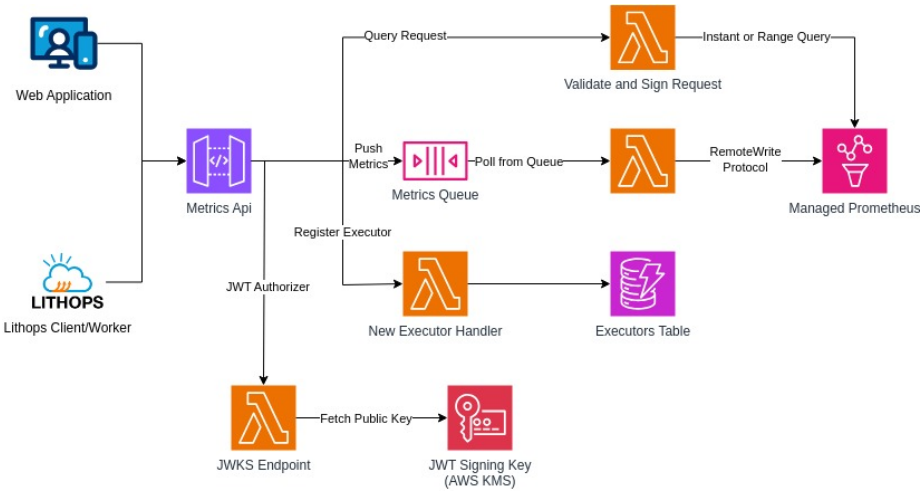

Figure 5: Architecture of the SaaS version of Lithops, with integration of AWS Managed Prometheus. The diagram illustrates how metrics are collected, validated, and securely stored in a scalable way using AWS managed services.

3.4 Profiler Implementation Details

3.4.1 Technologies and Tools Used

In the development of the performance monitoring system for serverless functions in Lithops, various technologies and tools were strategically selected to ensure the system's robustness. Below is a description of the technologies used and the justification behind their selection:

- **psutil**
  - **Description:** psutil is a Python library that provides access to system information related to processes and resource usage such as CPU, memory, disk, and network.
  - **Justification:** psutil was selected for its ability to provide real-time, detailed metrics with low overhead. Since the monitoring system is designed for serverless environments where resources are dynamically allocated, it was essential to use a tool that minimized performance impact and was compatible with diverse platforms. psutil allows efficient process monitoring without significantly affecting the execution of the monitored code.
  - **CPU Usage:** It is important to note that, according to the `psutil` documentation, CPU usage can exceed 100% in multi-core systems.

This is because CPU usage is measured relative to each available core, meaning that in a multi-core system, usage can exceed 100%. For example, in a four-core system, CPU usage of 400% would imply that all cores are being used at maximum capacity.

- **Prometheus**
  - **Description:** Prometheus is a time-series monitoring and alerting system designed to collect and query real-time metrics. It offers a flexible API for sending metrics and a query language (PromQL) for data analysis.
  - **Justification:** Prometheus was chosen for its ease of use, its ability to handle real-time data, and its integration with visualization tools like Grafana.
- **Tenacity**
  - **Description:** Tenacity is a Python library that implements automatic retry policies with exponential backoff mechanisms.
  - **Justification:** In distributed systems, temporary network failures are common. Tenacity was chosen to ensure the system's resilience by efficiently handling failures when sending metrics to Prometheus. With this tool, retry mechanisms were implemented to allow automatic system recovery, reducing data loss in cases of temporary failures.
- **AWS Managed Prometheus**
  - **Description:** AWS Managed Prometheus is an AWS-managed service based on Prometheus, which provides a scalable solution for collecting and storing metrics in cloud environments.
  - **Justification:** In the SaaS version of Lithops, AWS Managed Prometheus was selected for its ability to efficiently manage large volumes of metrics without the need for manual maintenance. Its automatic scalability makes it ideal for multi-user environments.

3.4.2 Development Process

The development of the monitoring system in Lithops was a process focused on robustness, scalability, and adaptability. Throughout the development, various technical challenges were addressed, which were resolved by implementing advanced techniques and using appropriate tools. Below is a detailed breakdown of the development process, divided into the design, coding, testing, and deployment phases.

Design

- **Description:** The design of the Lithops Profiler began with a thorough analysis phase aimed at gaining a deep understanding of the framework's architecture. A detailed breakdown of Lithops components was carried out, identifying their responsibilities and relationships. This understanding was essential to determine the optimal point where the Profiler should be integrated, so it could work harmoniously with other system components and support the execution of multiple workers in parallel.

- **Component Analysis:** During the initial days, key components such as the *Handler*, *JobRunner*, and *Invoker* of Lithops were carefully analyzed. The way each of these modules handled task distribution and worker communication was evaluated. This analysis allowed us to identify that the Profiler should be strategically created alongside the *Handler* to effectively capture performance metrics without interfering with the core task execution logic.

- **Modular Design and Scalability:** With the knowledge acquired, a modular Profiler was designed, capable of being integrated as an independent module within Lithops. This modular approach allowed the Profiler to be activated and deactivated based on user needs without impacting the base functionality of the framework. Additionally, it was designed so that each running worker could operate with its own Profiler instance, ensuring that the system could scale horizontally without bottlenecks in metric collection, as discussed earlier.

- **Initial Prototype:** Once the design was defined, a simple Profiler prototype was developed to validate its integration with Lithops. This prototype included basic metric collection capabilities, such as CPU usage. During this phase, tests were conducted to confirm that the Profiler could effectively scale along with the workers without causing significant system overhead.

- **Validation and Preparation for Full Development:** Before moving on to full development, design reviews were conducted to ensure that all aspects of the system, from metric collection to storage and visualization, were consistent with the performance and usability objectives. A detailed roadmap was prepared for development, including key milestones such as full integration with the Lithops backend and the creation of user documentation to facilitate the Profiler's adoption.

- **Iteration and Refinement:** After the prototype validation, the Profiler's design was iteratively refined. More metrics were added, exception handling was improved, and integration with *Prometheus Pushgateway* for real-time metric storage was implemented. Adjustments were made to optimize performance, including setting collection intervals and efficiently managing concurrency between multiple Profiler instances.

- **Use of Managed Prometheus for Run Lithops Cloud:** During the development and testing of the Profiler, it was identified that self-managed Prometheus could not efficiently scale in high-concurrency environments, resulting in significant overhead that affected system performance. To resolve this issue, AWS Managed Prometheus was adopted, offering an optimized and scalable infrastructure for real-time metric collection. This decision allowed for stable and efficient performance in environments with large numbers of serverless functions, eliminating the scalability issues observed earlier.

- **Security Considerations:** During the design phase, security aspects were also considered, particularly for implementation in *Run Lithops Cloud*. Access control mechanisms were designed to ensure that the metrics collected by the Profiler were accessible only by authorized users, respecting data privacy and security in a multi-tenant environment.

Coding

- **Description:** The coding phase followed an iterative and agile approach, where small, manageable increments of functionality were developed, tested, and refined continuously. This approach ensured that each new feature was smoothly integrated into the system without introducing regressions or performance issues.

- **Technical Challenges:**

  - **Minimizing Overhead:** One of the main challenges during coding was minimizing the overhead introduced by the Profiler. Since the Profiler had to operate in real-time, it was crucial that it did not significantly interfere with the performance of the applications being monitored. The metric collection interval (`profiler_timeout`) was optimized to ensure that metrics were collected at appropriate intervals without overloading the system. Additionally, a `random_delay` was implemented to stagger metric collections across different workers. This helped prevent multiple Profiler instances from sending metrics simultaneously, which could cause system load spikes.
  - **Handling Terminated Processes:** Another important challenge was handling processes that unexpectedly terminated. The `psutil.NoSuchProcess` exception was used to capture and handle these situations without interrupting the overall operation of the Profiler. This ensured that the system could continue collecting metrics from other active processes, maintaining the stability and integrity of the collected data.

Testing

- **Description:** Testing was crucial to ensuring that the Profiler could reliably operate across a wide range of scenarios, from local development environments to cloud deployments with thousands of workers. Unit, integration, and stress tests were conducted to validate both the functionality and efficiency of the system.

- **Technical Challenges:**

  - **Scalability Testing:** Simulations were conducted involving multiple workers running parallel functions, with the Profiler collecting and sending metrics to Prometheus in real-time. Initially, it was thought that the Profiler's overhead was significant due to the increase in overhead as more functions ran in parallel. However, after detailed investigation, it was determined that the Profiler actually had minimal overhead. The real issue lay with Prometheus, which did not scale adequately under intensive workloads. These tests confirmed that, despite the initial hypothesis, the Profiler could handle large volumes of data without compromising system performance, ensuring its ability to scale efficiently in serverless environments.

    I would like to highlight this challenge since it was discovered that the overhead increased with workloads involving hundreds of functions running in parallel. In certain workloads with up to 200 functions, the overhead was acceptable; however, beyond that number, the overhead began to increase significantly.

    After analyzing multiple logs and implementing debugging mechanisms in the Profiler, everything indicated that a process was not terminating correctly. Therefore, the investigation focused on thoroughly reviewing all possible factors that could cause process interruption.

    Once this possibility was ruled out, implementation and testing of a version using Managed Prometheus continued. In this version, regardless of the number of functions running, the overhead always remained minimal. This led to the conclusion that the issue lay in the scalability of the self-managed version of Prometheus, which was an initial hypothesis not confirmed until this problem was encountered.

    Therefore, the decision was made to implement the version with Managed Prometheus to ensure the scalability of Run Lithops Cloud. This is because, in Run Lithops Cloud, we can manage the backend (and therefore Prometheus), unlike using Lithops in its native form.

    Additionally, native Lithops users were advised through documentation that a self-managed Prometheus server could face scalability issues under intensive workloads. If users encounter this problem, they are recommended to create a workspace in Managed Prometheus or similar services to mitigate it.

    This issue will be the next step in ensuring proper monitoring in any version of Lithops. A possible solution will be to connect multi-

ple Prometheus nodes in a federated setup and complement it with Thanos to improve scalability, ensure long-term storage, and provide unified queries, which would efficiently manage large volumes of metrics in any Lithops environment.

  - **Resilience Testing:** Tests were conducted to validate the system's ability to automatically recover from temporary network failures and other infrastructure issues. These tests ensured that the Profiler could retry failed metric submissions and continue operating reliably, even in adverse conditions.

Deployment

- **Description:** The deployment of the monitoring system in Lithops was carried out in two main environments: the open-source version of Lithops and *Run Lithops Cloud*. It is important to note that the Profiler, being integrated into Lithops' codebase, does not require a special deployment. Once users install or upgrade to a new version of Lithops, the Profiler is automatically available for use.

- **Automated Configuration:** One of the key advantages of the Profiler is that it does not require any specific configuration for its operation. Since Lithops already sent some basic statistics to *Prometheus Pushgateway*, the Profiler's implementation simply extends this mechanism. As long as the log level (`log_level`) is set to DEBUG, the Profiler is automatically activated and begins collecting and sending detailed metrics to Prometheus. This means that users can continue using their existing Prometheus configuration without needing to make additional adjustments for the Profiler to function correctly.

- **Automation in Run Lithops Cloud:** In the SaaS version of Lithops, the configuration of Prometheus and related services was automated. This automation allowed users to deploy the monitoring system with predefined configurations, reducing deployment complexity and minimizing manual intervention. This was crucial to ensure that users could quickly start using the Profiler with minimal configuration.

- **Comprehensive Documentation:** To facilitate the use of the Profiler, detailed documentation was developed, including step-by-step guides for configuring and using the monitoring system. This documentation covers everything from basic installation to advanced configurations, ensuring that users can fully leverage the Profiler's capabilities in their own environments.

This deployment phase was thus designed to maximize ease of use and minimize entry barriers, ensuring that both open-source and Run Lithops Cloud users could benefit from the advanced monitoring capabilities without additional complications.

3.4.3 Integration Process

The integration process of the Profiler into Lithops was designed with a rigorous and methodical approach, employing software design principles that ensure modularity, cohesion, and maintainability. The integration was structured so that each system component functions independently but is efficiently connected to the others, forming a cohesive and robust system. Below is a detailed description of how this integration was achieved, supported by design methods and patterns.

Modular Design

- **The principle of Separation of Concerns** was central to the design of the Profiler. Each system component, represented by a specific class, was designed with a clearly defined responsibility. This approach not only facilitates the independent development and testing of each component but also reduces complexity and improves code maintainability.
- **Metric Collection Components:** Classes such as `CPUMetricCollector`, `MemoryMetricCollector`, `DiskMetricCollector`, and `NetworkMetricCollector` are responsible solely for collecting specific metrics. This modular design allows each of these collectors to be implemented and optimized independently without affecting other system components.
- **Metric Coordination and Management:** The `MetricCollector` class acts as a central container that organizes and manages the metrics collected by the different collectors. This design ensures that metrics, although coming from diverse sources, are handled in a uniform and consistent manner.

Composite Design Pattern: Metric Organization and Aggregation

- **The Profiler uses the Composite Design Pattern,** where `MetricCollector` plays the role of a composite object that aggregates and manages instances of the collected metrics. This pattern is well-suited to represent the hierarchy of metrics in a system where multiple collectors contribute to a larger, more complex data collection.
- **Metric Aggregation and Organization:** `MetricCollector` collects metrics from various collectors and stores them in organized lists (`cpu_metrics`, `memory_metrics`, etc.). This approach keeps the metrics organized and easily accessible by type or sequentially, facilitating further analysis and processing.
- **Unified Interface for Updates:** The `update` function in `MetricCollector` allows the integration of new collected metrics, ensuring that the metrics are synchronized and kept up-to-date at all

times. This design promotes consistency and makes it easier to extend the system with new types of metrics in the future.

Cohesion through Well-Defined Interfaces

- **The integration of the different components** of the Profiler was achieved through well-defined interfaces, ensuring that each component could interact with others without needing to know internal implementation details. This approach follows the Principle of Abstraction, which is key to maintaining cohesion and reducing dependencies between modules.
- **Metric Collectors and MetricCollector:** Methods such as `collect_metric` and `collect_all_metrics` define how metrics are collected and aggregated. These interfaces allow `MetricCollector` to communicate with the various collectors without needing to know how each type of metric is implemented.
- **Orchestration of the Monitoring Process:** The `Profiler` class uses these interfaces to orchestrate the collection and transmission of metrics. The `start_profiling` function centralizes the process, calling `collect_all_metrics` to gather data and then invoking `send_metric_to_prometheus` for transmission. This design ensures that the Profiler acts as a conductor, integrating all components into a coherent and controlled workflow.

Decorator Design Pattern: Exception Handling and Resilience To ensure system resilience and effectively handle errors, an approach based on the Decorator Pattern was implemented. This pattern allows extending the behavior of the `Profiler` classes by adding specific functionalities, such as exception handling or retries, without modifying the core logic.

- **Exception Handling via Decorators:** The `Profiler` classes are wrapped in decorators that capture and manage exceptions. For example, if a process terminates unexpectedly, the corresponding decorator intercepts this exception, ensuring the system continues to operate without interruption.
- **Exponential Retries with `tenacity` via Decorators:** Additional decorators implement exponential retries for metric transmission. These decorators apply automatic retries with increasing time intervals in the event of temporary failures, ensuring that critical operations are completed without manual intervention.

Minimizing Overhead: Efficient Configuration and Synchronization

- **The Profiler also incorporates mechanisms** to minimize overhead and optimize performance by efficiently configuring timing and synchronizing tasks.

- **Configuration of `profiler_timeout`:** The integration allows adjusting the time between metric collections via the `profiler_timeout` variable. This provides users with the flexibility to optimize Profiler performance based on the specific needs of their environment, allowing a balance between metric precision and resource usage.
- **Synchronization with `random_delay`:** To avoid load spikes when multiple instances of the Profiler run simultaneously, a `random_delay` was implemented. This technique staggers metric collections over time, reducing the likelihood of simultaneous operations overloading the system.

System Adaptability and Scalability

- **The modular design and integration based on clear interfaces** allow the Profiler to be adaptable and scalable across different environments, from local machines to distributed cloud infrastructures.
- **Adaptability to Different Environments:** The modularity allows users to configure the Profiler to collect different types of metrics and send them to various destinations (local or cloud), adjusting parameters such as collection interval and metric destination.
- **System Scalability:** Each Profiler instance operates independently, collecting metrics from specific processes without interfering with other processes or instances. This ensures that the system can scale efficiently, handling a large number of workers without creating bottlenecks or compromising the integrity of the collected data. In any case, the bottleneck lies with Prometheus, which will be addressed in the future improvements section.

Integration Validation and Testing

- **To validate that the Profiler integration functioned coherently and efficiently,** extensive integration tests were conducted:
- **Cohesion Testing:** It was ensured that all classes and methods in the Profiler worked together without conflicts, guaranteeing that metrics collected by different collectors were correctly integrated into `MetricCollector`.
- **Performance Testing:** The impact of the Profiler on the overall system performance was evaluated, ensuring that the overhead was minimal and that the synchronization of collections did not cause significant latency or load spikes.
- **Scalability Testing:** Simulations with a large number of concurrent workers confirmed that the system maintained its performance and accuracy in metric collection, demonstrating its ability to scale seamlessly.

This meticulous approach to the integration process ensures that the Profiler is not only effective and efficient but also highly adaptable and scalable to meet the demands of modern serverless environments.

3.5 Integration with Existing Systems and Platforms

The integration of the Profiler into Lithops has been designed to ensure compatibility with multiple cloud platforms and interoperability with other systems through efficient APIs. Below are the key aspects of this integration.

3.5.1 Compatibility with Cloud Platforms

The compatibility of the Profiler with multiple cloud platforms is essential for its deployment in diverse environments. This is achieved primarily through the use of cross-platform tools and libraries, as well as its direct integration into the Lithops codebase.

Use of `psutil` for Cross-Platform Compatibility One of the main strategies to ensure compatibility across different environments has been the use of the `psutil` library. As mentioned earlier, this library is responsible for collecting system metrics (such as CPU, memory, disk, and network usage) and is compatible with a wide range of operating systems, including Linux, Windows, and macOS. This allows for metrics collection without requiring specific adaptations for each operating system.

Direct Integration into Lithops Code The Profiler has been directly integrated into the Lithops source code, meaning it does not require complex additional configurations or separate deployments. This integration ensures that the Profiler is automatically available in any environment where Lithops is executed, whether in the cloud or on a local server. This strategy greatly simplifies the deployment process and reduces configuration errors, ensuring the Profiler works consistently across all environments supported by Lithops.

Compatibility Testing Extensive testing was conducted to ensure the Profiler works correctly on different backends such as localhost, Lambda, VMs, Kubernetes, and others. These tests confirmed that, thanks to the use of `psutil` and its direct integration with Lithops, the Profiler can operate seamlessly across any of these platforms.

3.5.2 API and Interoperability

The Lithops Profiler has been designed to efficiently integrate with other systems using standard APIs. In this context, interoperability is a key aspect achieved by adapting APIs to work in both open-source environments and the SaaS version of Lithops. Below are the most relevant aspects of this integration.

Access to Collected Metrics Access to the metrics collected by the Profiler is performed through Prometheus' native API. This widely used API ensures that metrics can be accessed and queried uniformly in different environments, whether using managed Prometheus (such as AWS Managed Prometheus) or a native Prometheus deployment.

- **Uniformity in Metric Access:** In both the open-source Lithops environment and the SaaS version, the metrics collected by the Profiler are available through the Prometheus API. This means that, regardless of whether a native or managed Prometheus deployment is used, the method of accessing the metrics is consistent, facilitating integration with existing monitoring and analysis tools.

API Evolution for Metric Submission Initially, Lithops had a simple API that allowed the submission of individual metrics to Prometheus Pushgateway. This API has been improved to allow for the submission of grouped metrics, optimizing efficiency and reducing network overhead.

API Versions It is important to note that there are two versions of this API tailored to the different environments in which Lithops is deployed. However, in both versions, the metrics provided and the configuration are the same; only the architecture differs, improving Prometheus' performance.

- **API for Lithops Open Source:** This version of the API is designed to work in Lithops' native open-source environment. In this case, the API uses standard Prometheus configurations to send grouped metrics. Users deploying Lithops in their own environments can access and configure this API to integrate it with their Prometheus instances.
- **API for Lithops SaaS (Run Lithops Cloud):** For the "Run Lithops Cloud" project, a SaaS platform based on Lithops, a specific version of the API has been developed. This version is optimized to function in a managed cloud environment, in this case, the backend of Run Lithops Cloud.

  Key Differences Between Versions: While the basic functionality of the API is maintained in both versions, the implementation in Lithops' SaaS environment includes additional measures to ensure that metrics are only accessible by authorized users in a shared environment. For this reason, the SaaS API version requires a token to authenticate requests.

  Advantages of the Run Lithops Cloud Version: This version offers a significant advantage in terms of scalability. Integrated with AWS Managed Prometheus, it ensures immense scalability that guarantees the proper collection and processing of all metrics, even in environments with

a large number of workers and massive data volumes. This allows the platform to efficiently handle large-scale workloads without compromising the accuracy or availability of the collected metrics.

Conclusion on Interoperability and API The evolution and adaptation of the Lithops Profiler API demonstrate a commitment to interoperability and efficiency across different environments. By improving the API to support grouped metric submissions and creating specific versions for open-source and SaaS environments, Lithops ensures that the Profiler can effectively integrate with other monitoring tools and systems, facilitating its adoption and use in various scenarios. This reinforces the Profiler's ability to operate efficiently and securely in large-scale production environments, both in the cloud and on-premises infrastructures, always ensuring the necessary scalability and accuracy for effective monitoring.

3.6 Performance Optimization and Scalability

Performance optimization and scalability are fundamental pillars in the design and operation of the Lithops Profiler. These aspects ensure that the system can handle variable workloads efficiently, adapting to user needs without compromising service quality.

3.6.1 Auto-Scaling Mechanisms

Implemented Auto-Scaling Mechanisms The Lithops Profiler is designed to operate in highly dynamic environments where workloads can fluctuate significantly. To manage these changes efficiently, auto-scaling mechanisms have been implemented to automatically adjust system resources based on demand.

- **Horizontal Scalability:** The Profiler in Lithops can scale horizontally, meaning that more Profiler instances can be added in response to increased workload. This is achieved by running multiple worker instances in parallel, each with its own Profiler that monitors and collects metrics independently.
- **Integration with AWS Managed Services:** In Run Lithops Cloud, AWS Managed Services' auto-scaling capabilities, such as AWS Lambda and AWS Managed Prometheus, are leveraged. These services allow the system to automatically scale resources based on usage metrics and workload, ensuring that the optimal amount of resources is always available without manual intervention.

  It is important to note that the Open Source version does not scale due to the nature of Prometheus.

Examples of Automatic Resource Adjustment

- **Workload-Based Scaling:** When an increase in the number of serverless functions needing monitoring is detected, the system can automatically deploy more Profiler instances to handle the increased metrics collection. For example, if a large number of workers are activated to execute concurrent tasks, more Prometheus instances can be deployed, or the capacity of AWS Managed Prometheus can be increased to handle the additional load without loss of performance.
- **Automatic Downscaling:** Similarly, when the workload decreases, the system can automatically reduce the number of Profiler instances or decrease the capacity of managed services, thereby reducing resource usage and optimizing costs.

3.6.2 Resource Management

Implemented Resource Management Strategies The monitoring solution in Lithops is designed to be resource-efficient without compromising system performance. Here are the strategies implemented:

- **Periodic Metric Collection:** Instead of continuously collecting metrics, the system can be configured to do so periodically and as described previously. This not only reduces CPU and memory load but also decreases network traffic and data storage, optimizing resource usage.
- **Efficient Thread Usage:** The Profiler uses psutil's threading capabilities to monitor multiple child processes of the JobRunner in parallel without the need to create additional processes. This allows for more efficient use of system resources by reducing the overhead typically associated with creating and managing multiple independent processes.
- **Optimized Metric Submission:** Instead of sending metrics individually, the Profiler batches collected metrics and sends them in groups to Prometheus. This approach reduces the number of necessary network requests, minimizing bandwidth usage and improving overall system efficiency.

These strategies ensure that the Lithops Profiler maintains an optimal balance between performance and resource usage, allowing for efficient operation in a variety of scenarios and deployment environments.

3.7 Security and Reliability Considerations

3.7.1 Security Measures

Implemented Security Measures Security in the Lithops monitoring system is a priority to protect the collected data and maintain system integrity at all times. The implemented measures include:

- **Data Encryption in Transit:** All communications between the Profiler and Prometheus are protected by TLS (Transport Layer Security) encryption. This ensures that sensitive data cannot be intercepted or altered during transmission.
- **Authentication and Authorization:** Access to the system and collected metrics is controlled through strict authentication and authorization mechanisms.
  - **IAM Roles:** In the SaaS version of Lithops, IAM (Identity and Access Management) roles are used to define which users can access which resources, ensuring that only users with the proper permissions can view or manipulate metrics.
  - **Bearer Tokens:** Each user receives a unique Bearer token, which must be presented with every request to the Prometheus API, ensuring that the request originates from an authorized user.
- **Prometheus API Security:** The Prometheus API used to access collected metrics is protected by authentication mechanisms. In the SaaS Lithops environment, this API is integrated with AWS Managed Prometheus, which includes additional security controls provided by AWS, such as automatic credential rotation.
- **Secure Exception Handling:** The Profiler is designed to handle exceptions securely. When errors are encountered during metric collection, such as the unexpected termination of a process, the system captures and handles these errors without compromising the security or integrity of the system.

3.7.2 Reliability and Fault Tolerance

System Fault Tolerance Features The reliability of the Lithops monitoring system is achieved through several fault-tolerance features, ensuring that the system remains operational and that data is collected and stored securely, even in the event of temporary failures or disasters.

- **Exponential Backoff for Network Failures:** In cases of network failures or when the connection to Prometheus is temporarily unavailable, the system uses the Tenacity library to implement exponential backoff retries. These automatic retries ensure that metrics are sent correctly after increasing intervals. If retry limits are reached, the metrics are discarded to maintain the integrity of real-time data (see Figure 6).

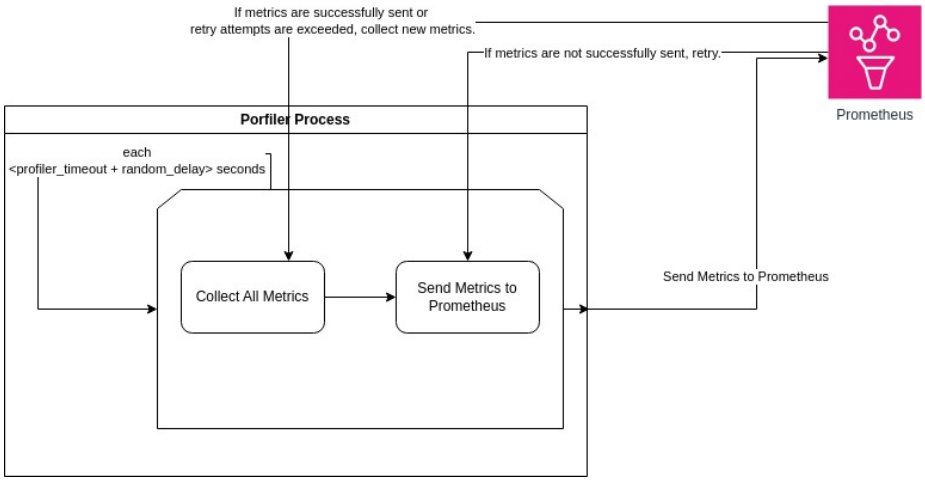

Figure 6: Exponential backoff retry mechanism using Tenacity.

- **Critical Exception Handling:** The system is designed to handle critical exceptions in a way that the metric collection process is not interrupted by unexpected process termination or system failures. Exceptions related to processes and metric submission are captured and handled without halting metric collection for other processes.

Strategies for Ensuring High Availability and Disaster Recovery

- **Redundant Deployment:** In Run Lithops Cloud, the monitoring infrastructure is deployed redundantly across multiple AWS availability zones. This ensures that even if one availability zone fails, the system can continue operating without interruptions.
- **Automatic Recovery:** In case Prometheus fails, the Profiler, using the Tenacity library, attempts to retry metric submission several times over a predefined time range. If, after these attempts, metrics cannot be sent, they are discarded to avoid accumulating outdated data. Once Prometheus becomes operational again, the Profiler resumes sending real-time collected metrics, ensuring that the metrics always reflect the current system state and avoiding the introduction of outdated data into the monitoring system.

These security and reliability measures ensure that the Lithops monitoring system is robust and prepared to operate securely in various environments.

3.8 Validation and Testing

3.8.1 Testing Methodologies

Overview of Testing Methodologies The validation and testing of the Profiler in Lithops were carried out through a comprehensive and structured approach to ensure the system's robustness, reliability, and performance. The testing process was meticulously planned to cover all aspects of the Profiler's functionality, from individual components to the fully integrated system.

Unit Testing Unit tests were the first step in the testing process. Each class and function within the Profiler, such as `CPUMetricCollector`, `MemoryMetricCollector`, `DiskMetricCollector`, and `NetworkMetricCollector`, was tested in isolation to verify that they behaved as expected under various conditions. *Mocking* techniques were used to simulate different system states, allowing for the creation of controlled test environments. For example, calls to `psutil` were mocked to return predefined values, allowing for the verification of metric collection logic without relying on real system metrics.

Integration Testing Integration tests focused on ensuring that the various components of the Profiler worked together seamlessly. This phase included testing the interaction between `MetricCollector` and individual metric collectors, as well as the integration of the Profiler with the Lithops framework. These tests verified that metrics were collected, aggregated, and transmitted to Prometheus accurately. Additionally, it ensured that the Profiler did not interfere with the normal execution of serverless functions, maintaining the overall performance and reliability of the system.

Performance Testing Given the distributed nature of serverless environments and the potential for high concurrency, performance testing was crucial. The Profiler was subjected to scenarios with thousands of concurrent serverless functions to assess its ability to handle large-scale operations without introducing significant overhead. Performance tests focused on measuring system resource consumption (CPU, memory, and network) under high load conditions, ensuring that the Profiler remained efficient and did not become a bottleneck.

Fault Tolerance Testing Fault tolerance tests were conducted to evaluate the Profiler's resilience in the face of failures, such as process terminations, network interruptions, or system failures. The testing process simulated various failure scenarios, such as the abrupt termination of monitored processes or network disconnection during metric transmission. The goal was to ensure that the Profiler could handle these failures efficiently, either by retrying operations or shutting down safely without data loss. The system's ability to recover from failures and resume normal operation without manual intervention was a key focus of these tests.

End-to-End Testing End-to-end tests involved deploying the Profiler in a full Lithops environment, simulating real-world workloads. These tests validated the complete data flow, from metric collection and aggregation to transmission and visualization in Prometheus. End-to-end testing was crucial to verify that the system met overall requirements and functioned correctly in an integrated environment, with all components interacting as expected.

Security Testing Security tests were conducted to ensure that the Profiler adhered to best practices for data protection, access control, and secure communication. Tests were performed to verify that metrics were transmitted securely to Prometheus, unauthorized access was prevented, and data integrity was maintained. Security testing also evaluated the effectiveness of encryption, authentication, and authorization mechanisms in protecting sensitive information.

3.8.2 Testing Execution

Approach for Test Execution In this thesis, unit and integration tests will be conducted using a custom test pipeline that allows detailed control over the execution and verification of the collected metrics. This pipeline has been specifically designed to execute tests sequentially, allowing each Profiler component to be evaluated in isolation before proceeding to the full system integration.

Verification of Results and Collected Metrics The verification process will be carried out through various pipelines, each focusing on specific areas:

Test 1: Fibonacci (1 stage) The first test will focus on unit tests, where each class and function of the Profiler will be evaluated individually. In this phase, the tests will simulate different system states, verifying that the collectors' metrics reflect the expected and correct behavior.

In this test, a `localhost` backend will be used, launching 10 functions that calculate the Fibonacci number of 44.

```python
import time
from lithops import FunctionExecutor

def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

def main():
    with FunctionExecutor(log_level='DEBUG') as fexec:
        fib_numbers = [42] * 10
        futures = fexec.map(fib, fib_numbers)
        results = fexec.get_result(futures)

    for result in results:
        print(result)

if __name__ == "__main__":
    main()
```

Listing 1: Test code for Fibonacci calculation.

Results Analysis with Grafana Dashboards

CPU The graph in Figure 7 shows fluctuations in CPU usage, with peaks up to 120% and lows around 90%. This behavior reflects the computation-intensive nature of the Fibonacci algorithm, where high processing capacity is required.

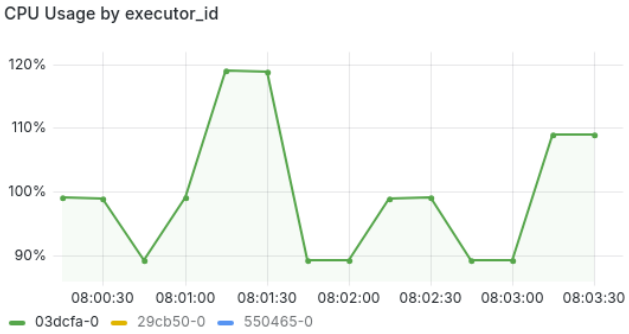

Figure 7: CPU usage by executor_id during Test 1.

Disk Figure 8 shows the disk usage metrics. In "Disk Read by executor_id," it is observed that no disk reads occurred during execution, which is

expected as the Fibonacci calculation does not require disk access.

Regarding writes, "Disk Write by executor_id" indicates only small writes (6 kB), likely related to system operations. The read and write rates ("Disk Read Rate" and "Disk Write Rate") remain close to 0 MB/s, confirming that disk operations were minimal, consistent with the computational nature of the executed code.

Figure 8: Disk usage metrics during Test 1.

Memory Figure 9 shows the memory usage metrics during the execution of the pipeline. Memory usage remains constant around 30 MB throughout the test, which is consistent with the expected behavior for a calculation that does not require large amounts of data or use complex memory structures.

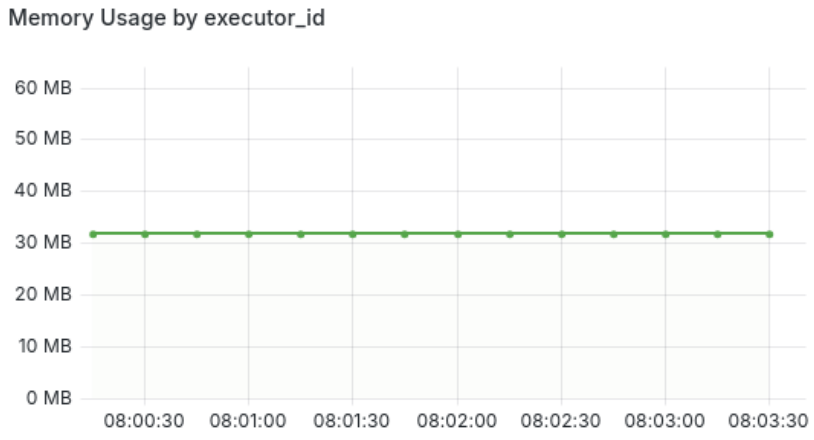

Figure 9: Memory usage metrics by executor_id during Test 1.

Network In this case, network usage metrics are ignored because `psutil` does not provide network metrics specific to the monitored processes, but rather reports the machine's total network usage. Since this is a local (localhost) backend, these metrics are not relevant as they include network usage from other unrelated tasks. However, in cloud backends like AWS Lambda, EC2, or Kubernetes, where the machine is exclusive to task execution, these metrics reflect actual network usage associated with our code, making them more useful in such environments.

Test 2: Fibonacci (multistage) In this test, the code will be the same as in *Test 1*, but during execution, we will introduce a 5-second `sleep` in the middle of the process. The purpose of this adjustment is to observe how resource usage behavior changes during the pause and to confirm that CPU, memory, disk, and network usage metrics adjust as expected. This modification allows us to validate whether the Profiler correctly captures idle periods and their impact on resource usage.

Additionally, this test will use the `AWS Lambda` backend along with the `Managed Prometheus` endpoint. In this way, we will be able to visualize the results through the *Run Lithops Cloud* platform, leveraging managed monitoring capabilities in the cloud.

The test will consist of four different stages:

- **Stage 1:** 400 functions calculating Fibonacci of 25.
- **Stage 2:** 100 functions calculating Fibonacci of 30.
- **Stage 3:** 25 functions calculating Fibonacci of 34.
- **Stage 4:** 4 functions calculating Fibonacci of 36.

Results Analysis with Run Lithops Cloud

Gantt The Gantt chart shows the detailed durations of concurrent tasks (Figure 10). We can see how tasks execute in parallel, with intense activity peaks followed by idle periods. The visual representation of "Active Calls" aligns with the executions of the different stages, verifying the expected behavior of concurrency during execution.

The lilac-colored fragments correspond to "Other Times," times that do not belong to System Time, User Time, or Upload Time. It is notable that in many functions, especially in the first stage (which has more functions running in parallel), these Other Times are present. This behavior is due to the backoff strategy in retries for sending metrics to Managed Prometheus. Even with scalable solutions like Managed Prometheus, some overhead can be experienced when working with high levels of concurrency. However, in the following stages, as observed, this overhead is minimal.

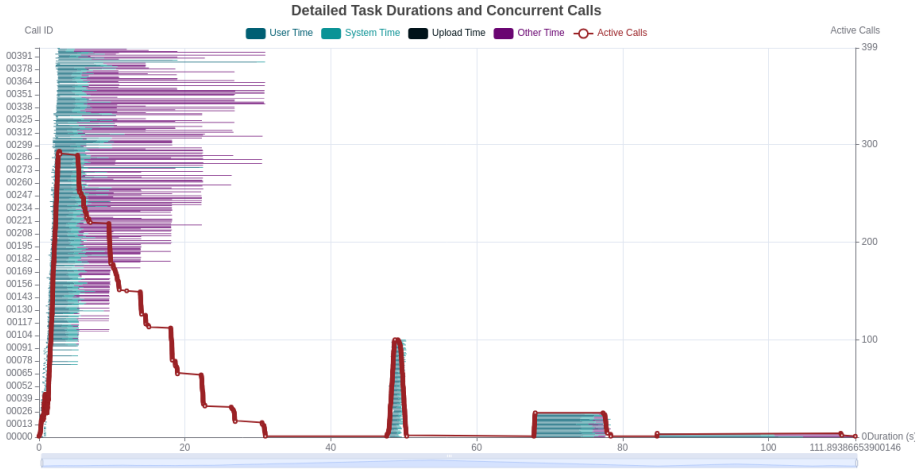

Figure 10: Gantt chart showing task durations for Test 2.

CPU The CPU usage graph (Figure 11) reflects a progressive increase in resource consumption as the stages progress. More complex functions, such as those calculating higher Fibonacci numbers, show more intense CPU usage towards the end of the process, reaching up to 100% average usage.

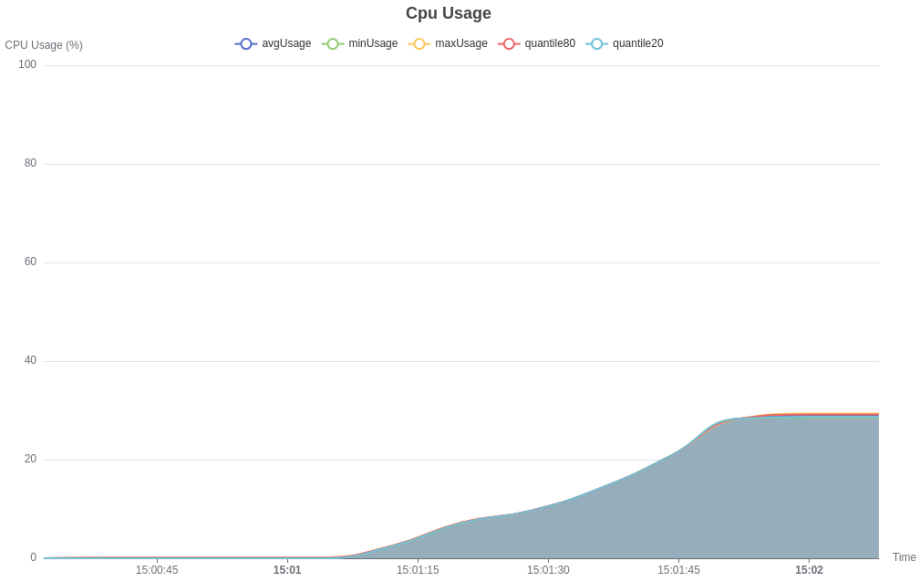

Figure 11: CPU usage during Test 2.

Disk In the disk usage analysis (Figure 12), both read and write operations show very low absolute values. Although peaks are recorded during the more complex calculation stages, the volume of data read and written is minimal, reflecting that the execution of Fibonacci functions does not require intensive disk usage. The read/write rate (Figure 13) follows a similar pattern, with slight increases in write operations when storing the function results, though the values are practically insignificant.

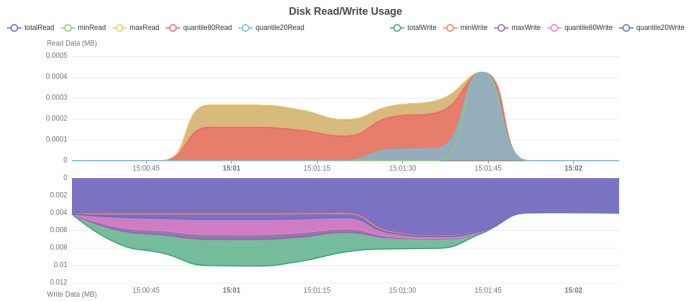

Figure 12: Disk usage during Test 2.

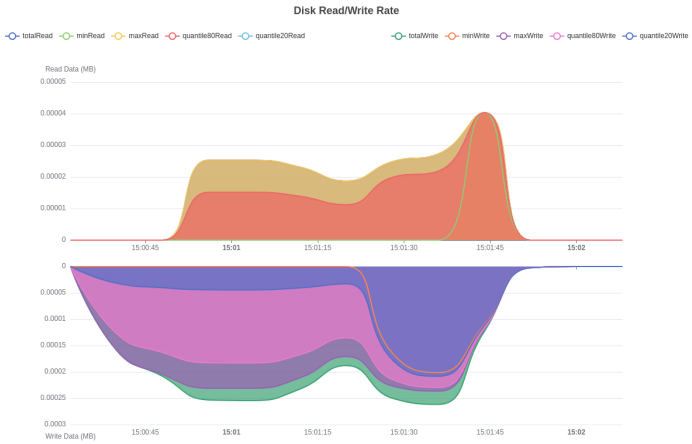

Figure 13: Disk read/write rates during Test 2.

Memory The memory usage graph (Figure 14) shows a steady increase during the execution of more complex tasks, reaching around 120 MB. This is consistent with the increasing complexity of the Fibonacci calculations in the later stages.

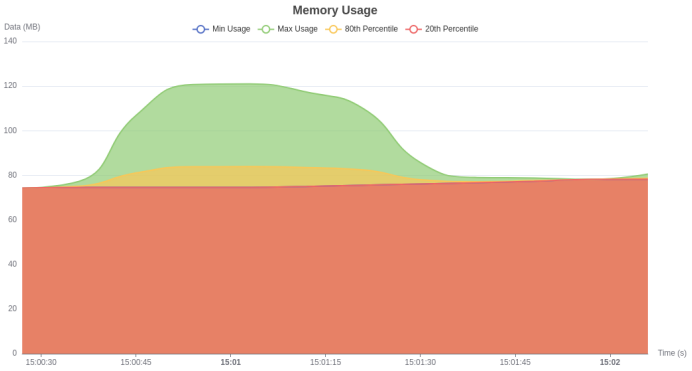

Figure 14: Memory usage during Test 2.

Network In the case of network usage (Figure 15), moderate read and write activity is observed, corresponding to communication between workers and the AWS Lambda service. The data transfer rate (Figure 16) follows a similar pattern, with brief increases during the transmission of results and requests.

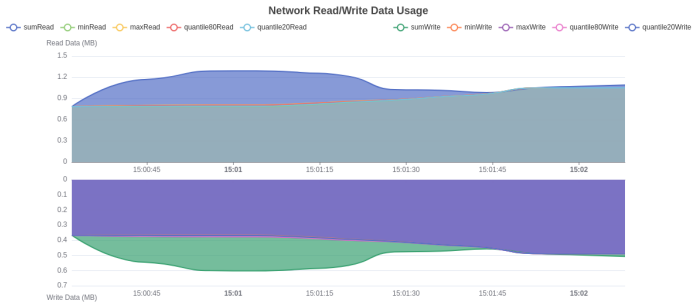

Figure 15: Network usage during Test 2.

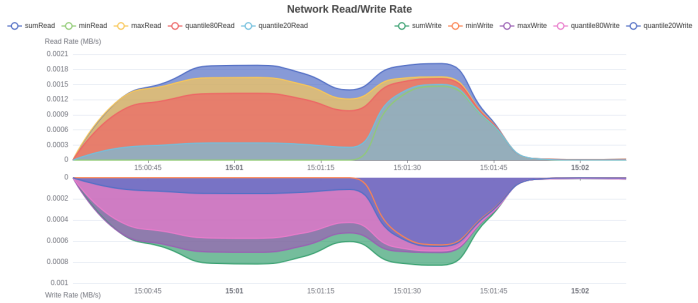

Figure 16: Network read/write rates during Test 2.

These graphs confirm that the Profiler accurately captures resource usage metrics, and that the overall system behavior during executions matches expectations, both during intensive calculation periods and idle times.

Test 3: Geospatial Data Processing for Evapotranspiration Calculation (split size 4 with 25 files) This pipeline focuses on a geospatial use case where evapotranspiration rates (ETC) are calculated for different geographic areas using the **Penman-Monteith method**. The primary objective is to process large volumes of geospatial data, including **Digital Terrain Models (DTMs)**, along with climate data such as temperature, humidity, wind speed, and solar radiation, to determine the water consumption by crops in the **Murcia region of Spain**.

Workflow The pipeline is divided into several stages that handle data preparation, interpolation, and evapotranspiration calculation:

- **Data Preparation**:
  - **Digital Terrain Models (DTMs)**: Input files in ASCII or GeoTIFF formats are converted into **Cloud Optimized GeoTIFFs (COG)** for optimized performance.

  - **Climate Data (SIAM)**: Meteorological data, including temperature and humidity, is uploaded from a CSV file into an object storage system (AWS S3).
  - **Shapefile of Murcia**: A shapefile containing the boundaries of the Murcia region is uploaded for delineating the areas of analysis.

- **Data Interpolation**:
  - The terrain is divided into subtiles to increase parallelism. This enables large DTMs to be broken into smaller chunks for processing in parallel on AWS Lambda.
  - Solar radiation is calculated for each subtile using geospatial processing libraries such as **GRASS**. Additionally, climatic variables (temperature, humidity, wind) are interpolated using **Inverse Distance Weighting (IDW)**, generating raster maps for each variable.

- **Evapotranspiration Calculation**:
  - The **Penman-Monteith equation** is used to combine the interpolated maps of temperature, wind, humidity, and solar radiation, along with a crop coefficient (Kc), which adjusts based on the crop type (vineyards, olive groves, fruits, etc.).
  - Evapotranspiration is computed both globally for the entire region and specifically for each crop shape defined in the shapefile.

- **Output Generation**:
  - The pipeline generates raster maps representing the daily evapotranspiration (in mm) for each processed subtile. These results are stored back in the object storage and visualized using plots generated with **Matplotlib**.

Expected Results The primary objective of this pipeline is to efficiently process large volumes of geospatial data using **massive parallelism** and distributed computing on serverless environments. This pipeline provides information such as the number of processed files and the estimated cost of the execution. Additionally, the pipeline generates clear visual representations of the processed data, such as maps illustrating water consumption by crops in the region for a specific day.

By processing large geographic areas and generating precise evapotranspiration maps, this pipeline aims to optimize water usage in agriculture, improving sustainability in irrigation and crop management.

For more detailed information about this geospatial workflow, you can refer to the official repository [39].

Results Analysis with Grafana Dashboards In this pipeline, the number of splits used to divide the data into smaller blocks can be adjusted. The number of splits directly affects the number of tiles, as the number of tiles is equal to the square of the split value. This value controls the level of parallelization employed in the pipeline, which can influence the performance and efficiency of the process. For this test, a split size of 4 was used, resulting in a total of 16 tiles, thereby improving data processing parallelization. Additionally, a total of 25 input files with an input size of 0.25GB were used.

Gantt The Gantt chart (Figure 17) clearly shows the concurrent tasks and their duration. The fourth stage is the longest due to the nature of the code. In this case, we see that the overhead (other times) is almost nonexistent.

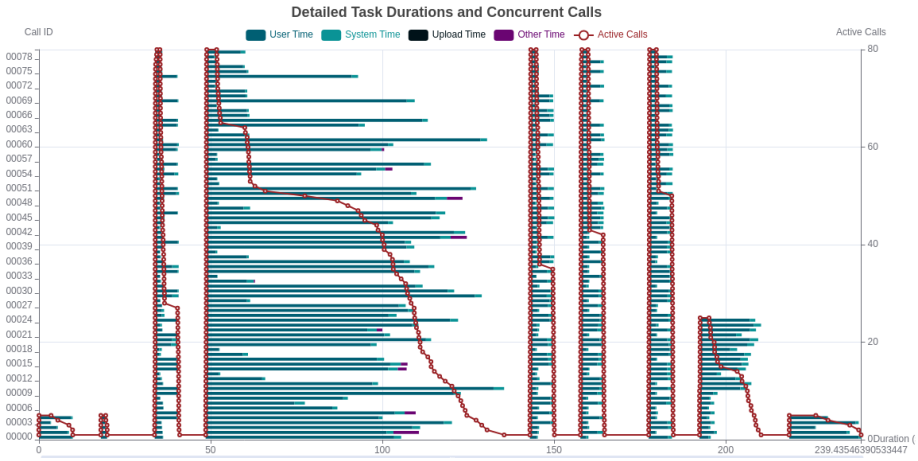

Figure 17: Gantt Chart - Pipeline 3

CPU The CPU usage graph (Figure 18) shows stable fluctuations during the most intensive processing phases, with peaks reaching 100%. Elevated usage is observed midway through the execution, which likely explains the fourth stage mentioned earlier, the longest of all. During the processing, the average usage fluctuates mostly between 30% and 80%, indicating good resource utilization without saturation.

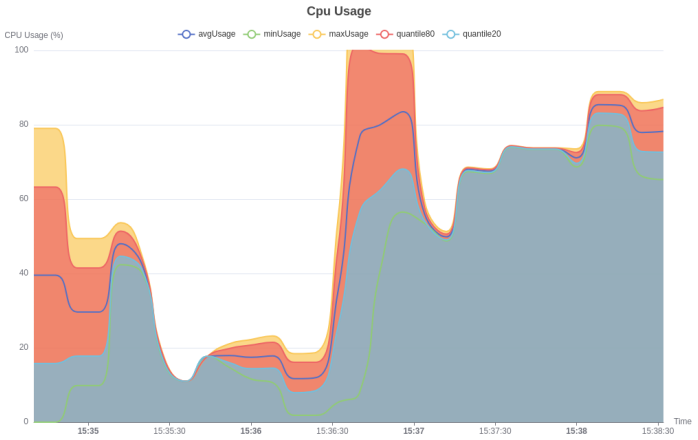

Figure 18: CPU Usage - Test 3

Disk Disk usage (Figure 19) shows peaks at the beginning, middle, and end of the process, coinciding with the input and output file read/write operations. The read/write rate (Figure 20) follows the same pattern, highlighting data input reads and output writes.

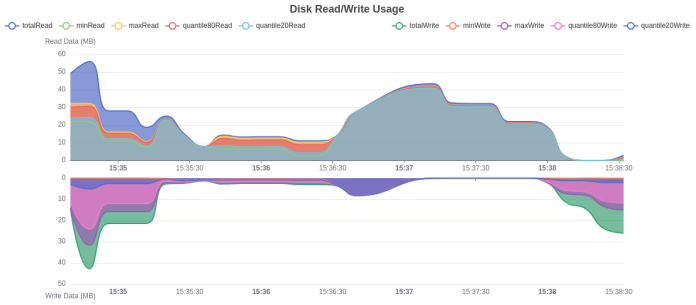

Figure 19: Disk Usage - Test 3

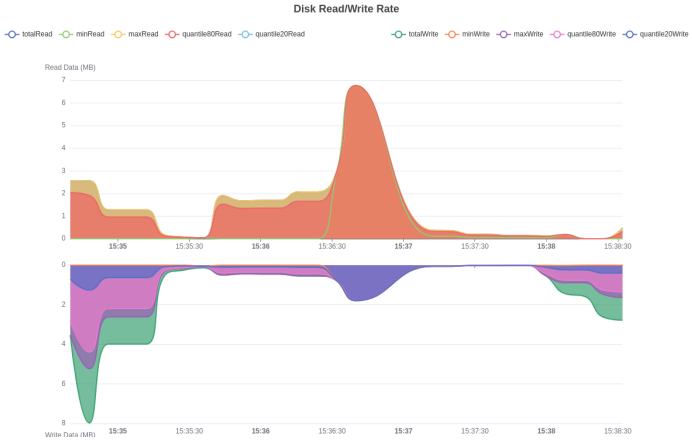

Figure 20: Disk Read/Write Rate - Test 3

Memory Memory usage (Figure 21) shows a gradual decrease after a high initial consumption, stabilizing during most of the process. At the end of the pipeline, another peak occurs, related to result storage.

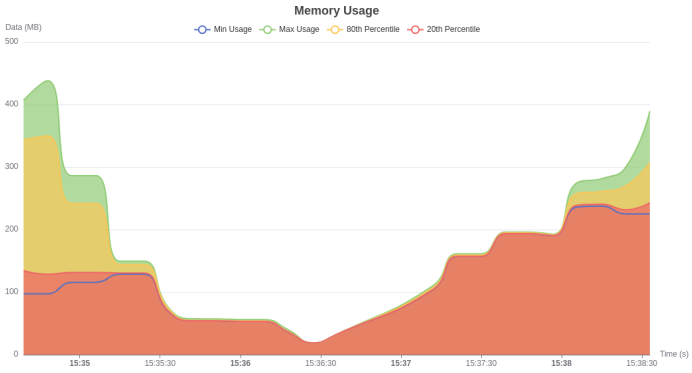

Figure 21: Memory Usage - Test 3

Network Network usage (Figure 22) is distributed throughout the pipeline, with notable peaks in the middle, corresponding to input/output file transfers. Regarding the transfer rate, it is naturally higher at the beginning and the end, as shown in Figure 23.

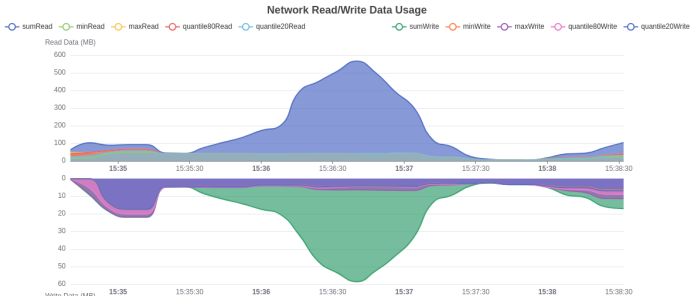

Figure 22: Network Usage - Test 3

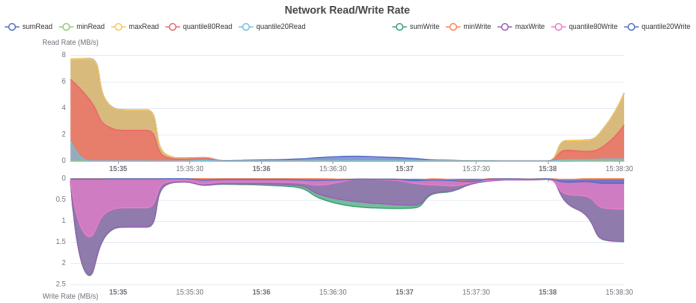

Figure 23: Network Read/Write Rate - Test 3

Test 4: Geospatial Data Processing for Evapotranspiration Calculation (split size 5 with 25 files) To compare with the previous test, another execution will be performed with the same parameters but with the split size changed to 5, resulting in a total of 25 tiles.

Results Analysis with Grafana Dashboards

Gantt The Gantt chart (Figure 24) shows how the number of executed functions is greater than in the third test. This is due to the split size change from 4 to 5.

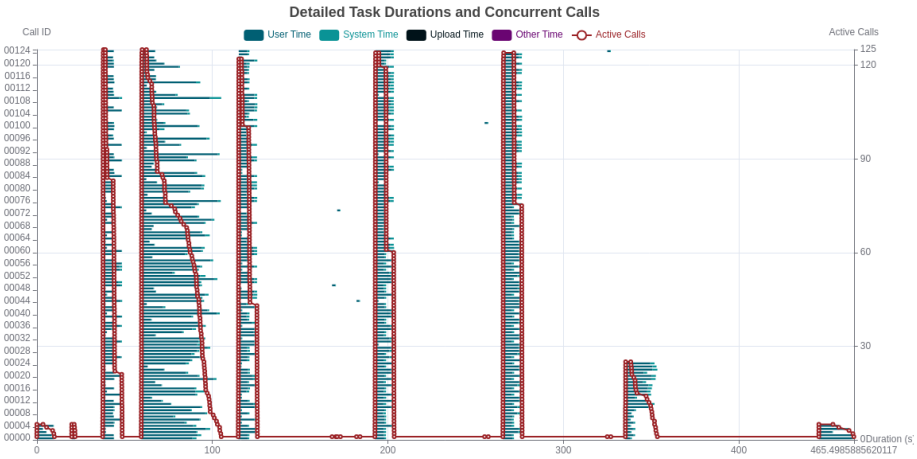

Figure 24: Gantt Chart - Test 4

CPU The CPU usage graph (Figure 25) allows us to differentiate how CPU usage rises and falls throughout each stage. This is interesting to compare with the previous test to conclude which one makes better use of resources.

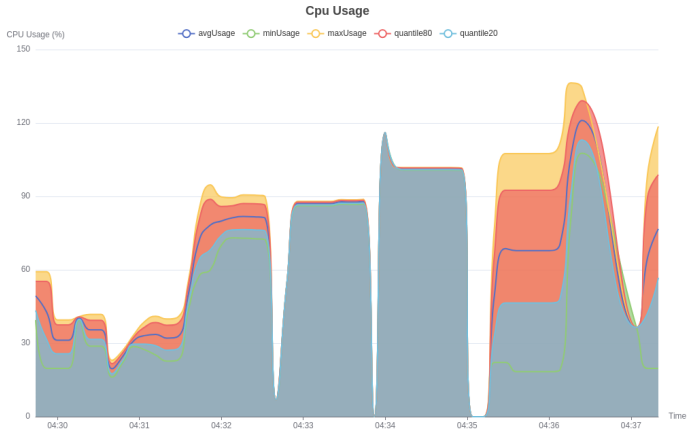

Figure 25: CPU Usage - Test 4

Disk Disk usage (Figure 26) shows the same behavior as in the previous test.

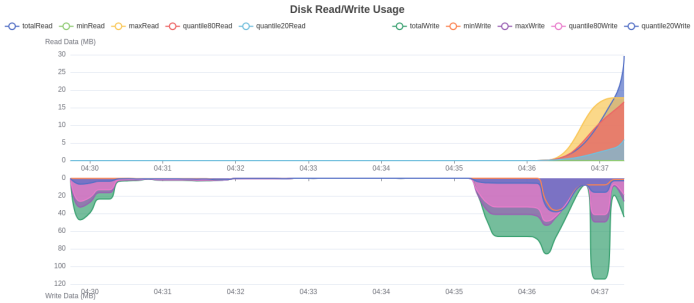

Figure 26: Disk Usage - Test 4

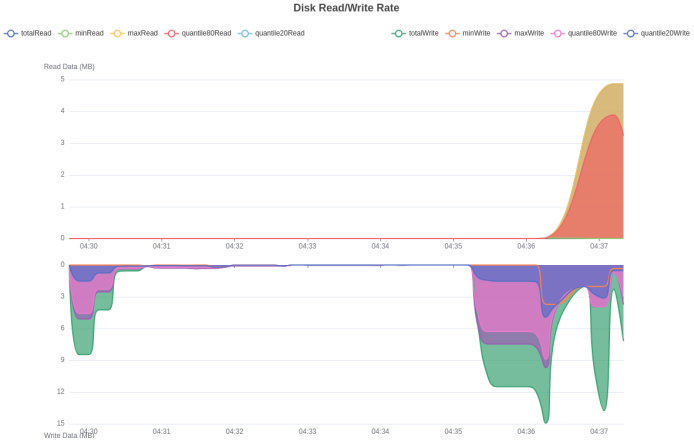

Figure 27: Disk Read/Write Rate - Test 4

Memory Memory usage (Figure 28) behaves as expected for this pipeline.

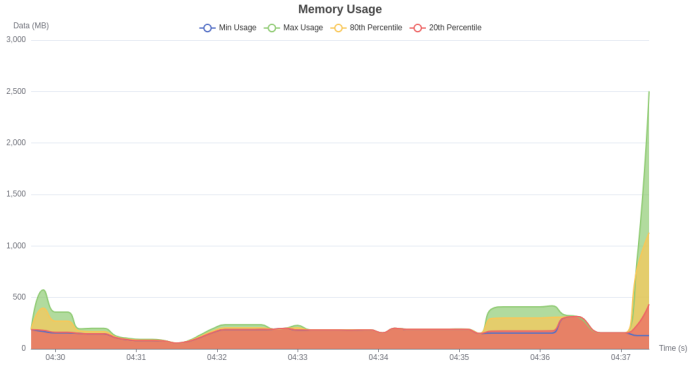

Figure 28: Memory Usage - Test 4

Network The network usage (Figure 29) and transfer rate (Figure 30) show similar patterns to the previous metrics.

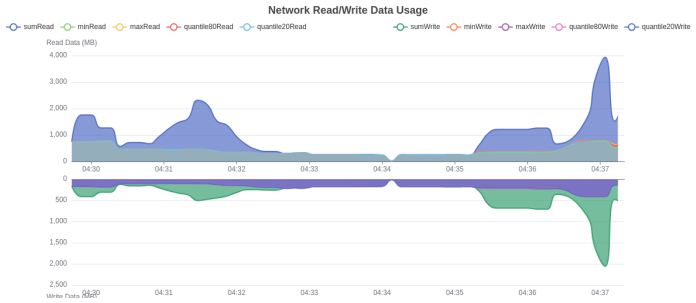

Figure 29: Network Usage - Test 4

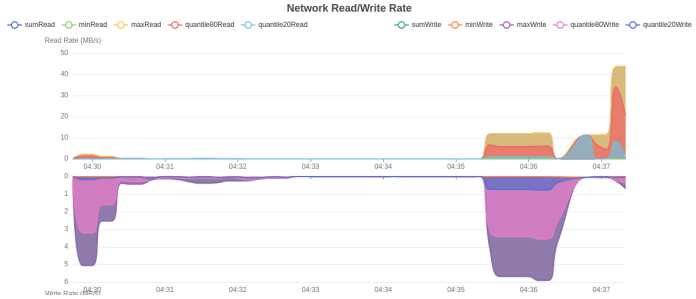

Figure 30: Network Read/Write Rate - Test 4

Bonus: Metabolomics Pipeline To conclude the tests, a brief presentation of the Gantt diagram (Figure 31) corresponding to the execution of a metabolomics pipeline developed with `Lithops` will be provided. This pipeline aims to process mass spectrometry data at scale for metabolomics analysis, leveraging the scalability and efficiency of serverless computing. The source code and more details on this pipeline can be found in the official `GitHub` repository [22].

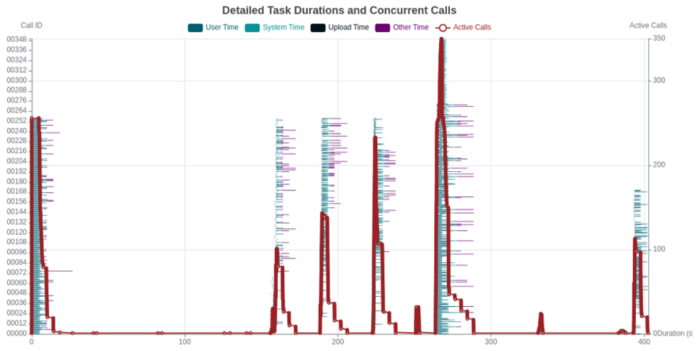

Figure 31: Gantt Chart of the Metabolomics Test

3.8.3 Test Conclusions

The results obtained from the different pipelines have validated the performance of the `Lithops` monitoring system in real-world scenarios, measuring resource usage such as CPU, disk, memory, and network. Below are the main conclusions:

1. Efficient Use of Resources: Across all the pipelines executed, CPU usage generally remained stable and aligned with expectations based on the computational load of the tasks. Although there were peaks that reached 100% usage, these moments coincided with computation-intensive phases, indicating that the resources were being well utilized. Similarly, memory usage charts showed consumption consistent with the operations of each pipeline, without any evidence of resource saturation.

2. Scalable Parallelization: One key element in the pipelines was the ability to adjust the level of parallelization by modifying the number of splits. As this value increased, so did the number of sub-tasks (or tiles), improving the distribution of the workload among the different workers.

3. Interaction with Storage: Network behavior was heavily influenced by read and write operations to cloud storage, as expected. This was clearly seen in the geoprocessing pipeline charts, where accessing large volumes of data is an integral part of the process.

4. Comprehensive Monitoring: The use of `Grafana` and `Prometheus` enabled detailed monitoring of all utilized resources. Real-time data visualization helped identify resource usage patterns and potential bottlenecks.

5. Pipeline Flexibility: The different pipelines tested demonstrated the flexibility of `Lithops` to adapt to various use cases, from simple numerical calculations like Fibonacci to complex geoprocessing workflows and metabolomics

analysis. This versatility, combined with the fine-tuning of parallelism and cloud data handling, reinforces the platform's power in distributed computing environments.

In summary, the tests showed that `Lithops` is a robust and efficient platform for running large-scale *serverless* applications, capable of effectively handling parallel task processing, resource usage, and cloud storage interaction. This makes it an ideal option for tasks requiring high performance and scalability, such as geospatial data processing and advanced scientific analysis.

3.9 Conclusions

The *Profiler* in `Lithops` has proven to be an effective and lightweight tool for monitoring *serverless* environments and other execution contexts. The tests conducted in various scenarios, such as local environments (*localhost*) and *serverless* platforms, have confirmed its versatility. Furthermore, the system has been tested both with self-managed `Prometheus` on a local server and with `Managed Prometheus`, demonstrating its adaptability to different monitoring configurations.

This versatile behavior allows jobs to be analyzed across multiple situations and scenarios, which is key to ensuring that the collected metrics are applicable in diverse environments. For users of the open-source version of `Lithops`, the metrics can be visualized using `Grafana`, while users of the *Run Lithops Cloud* platform can utilize the integrated dashboards, which provide clear and accessible real-time metric visualizations.

In terms of performance, the *Profiler* efficiently collects key metrics (CPU, memory, disk, network) with minimal impact on the pipelines. Its integration with `Prometheus` and `Grafana` facilitates bottleneck detection and resolution. Additionally, it adapts well to different levels of parallelism without negatively impacting function execution, reinforcing its scalability and flexibility.

The system also proved resilient, handling temporary failures without compromising the integrity of the collected data. Although there are potential improvements, such as optimizing latency in capturing network and disk metrics, the *Profiler* meets the monitoring requirements for distributed and *serverless* systems, providing a robust and flexible solution.

4 Metrics Visualization

This thesis provides the *Lithops Profiler* solution for both the open-source version of `Lithops` and the *Run Lithops Cloud* platform, each using different tools for metric visualization.

In the open-source version, users can set up a `Grafana` server to visualize metrics collected by `Prometheus`. `Grafana` offers customizable dashboards that display real-time metrics such as CPU, memory, and disk usage, providing a flexible, cost-effective monitoring solution for users who manage their own infrastructure.

For *Run Lithops Cloud*, dedicated dashboards have been created using `Apache ECharts`, a lightweight and integrated tool for real-time monitoring. `ECharts` simplifies the process by providing pre-configured dashboards directly within the platform, reducing setup complexity and optimizing resource use.

In summary, `Grafana` offers flexibility for open-source users, while *Run Lithops Cloud* provides a streamlined experience through integrated `ECharts` dashboards, ensuring effective metric visualization in both environments.

4.1 Grafana Dashboards

The use of `Grafana` for the open-source version of Lithops enables users to visualize metrics collected from various jobs and functions. The dashboards are designed to provide insights ranging from general performance overviews to detailed, granular metrics. Below are the created dashboards:

4.1.1 Generic Dashboard

The `Generic Dashboard` provides an overall view of the resources used by all the user's jobs. It aggregates metrics such as CPU, memory, and disk usage across all functions of the same job. Users can filter by specific `call IDs` or view metrics for all calls. Additionally, if the backend supports it, users can filter by worker instances, offering a broad perspective on resource consumption.

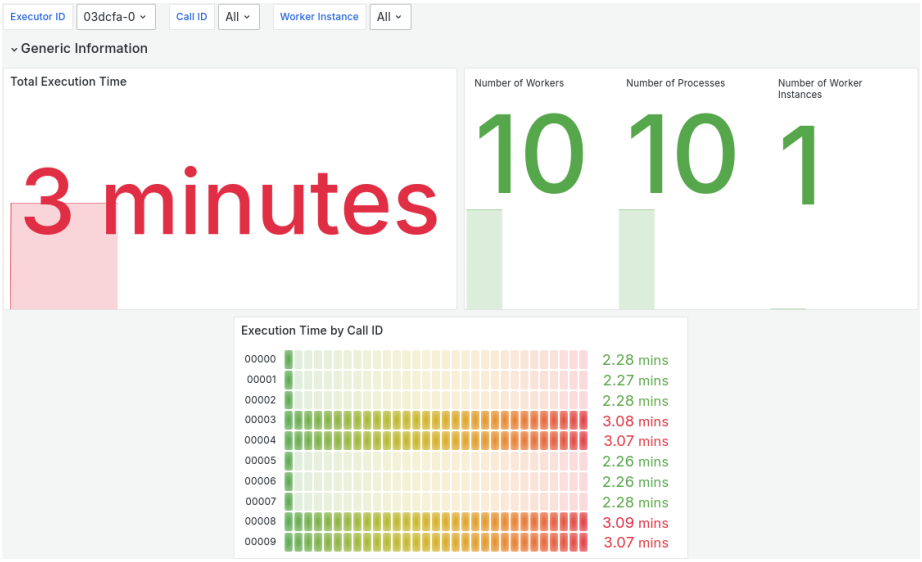

Figure 32: Generic Dashboard for Lithops Jobs - CPU, Memory, and Disk Usage.

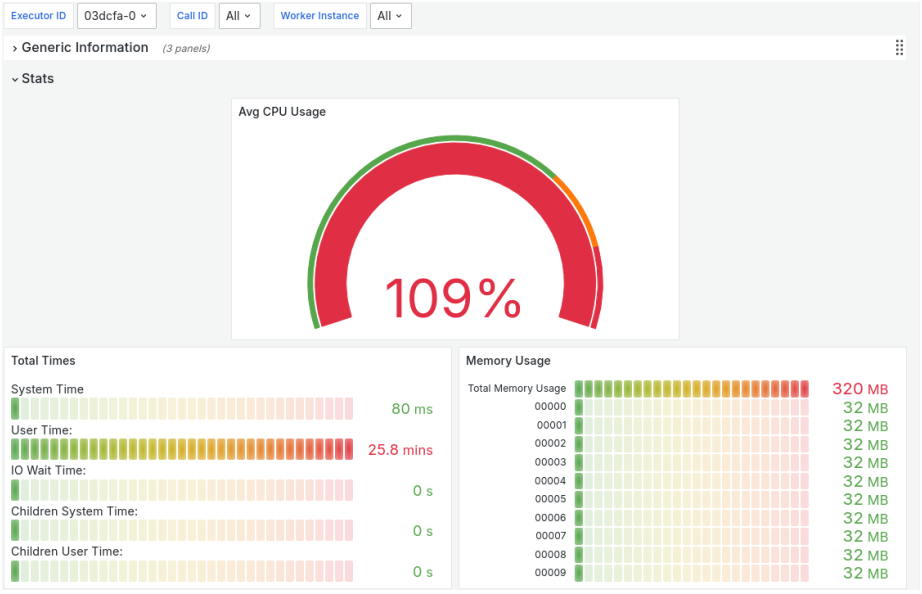

Figure 33: Detailed Metrics Visualization in the Generic Dashboard.

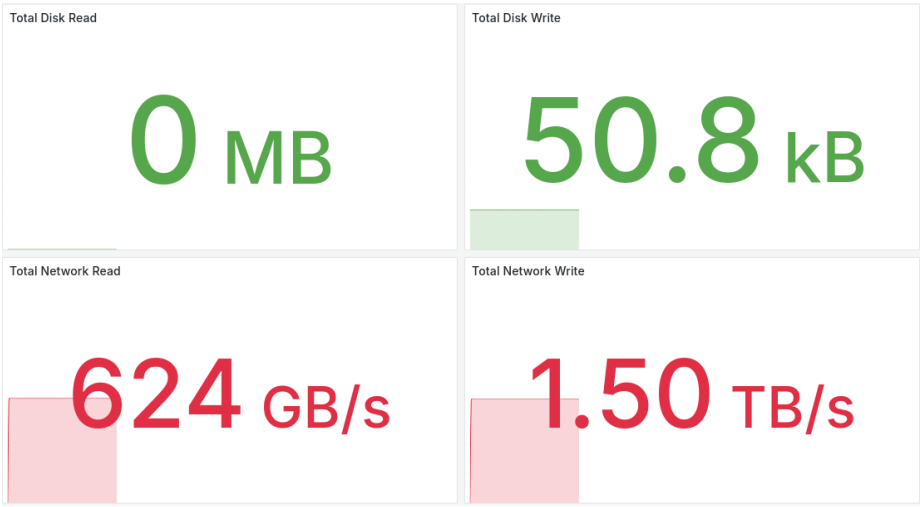

Figure 34: Aggregated Resource Usage Metrics in the Generic Dashboard.

4.1.2 General Performance Dashboard of the Executor

This dashboard aggregates performance metrics for each job executed by an `Executor`. It provides a job-level summary, showing metrics like overall CPU, memory, and execution time, making it ideal for users looking to monitor the efficiency and resource usage of their jobs at a higher level.

Figure 35: Overall Performance Dashboard for Executor Jobs.

Figure 36: Aggregated CPU and Memory Usage by Executor.

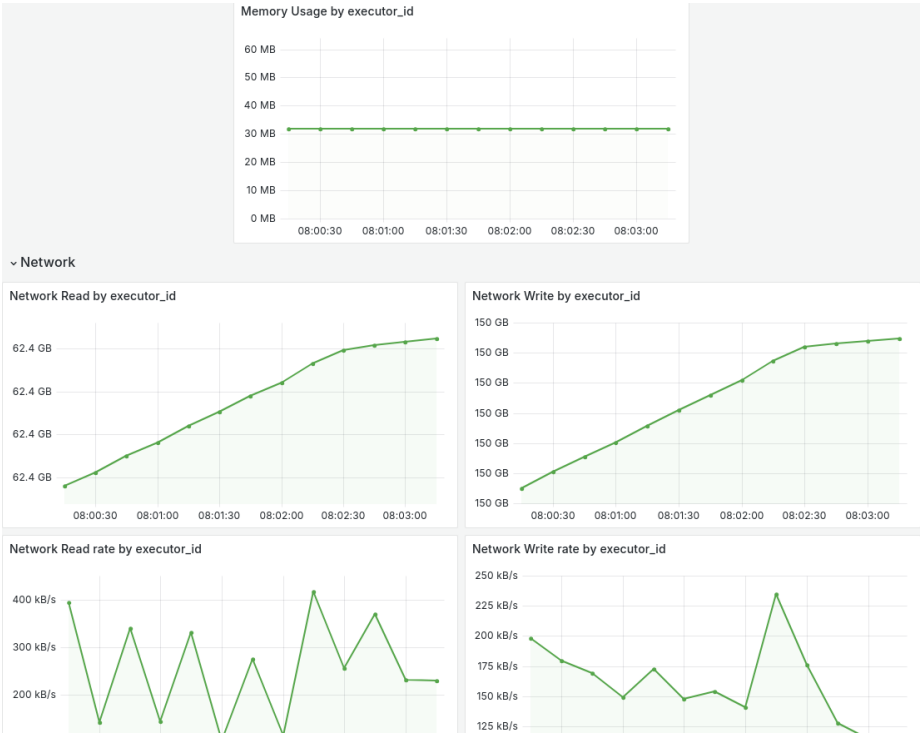

Figure 37: Execution Time and Resource Usage for Specific Executor Jobs.

4.1.3 Detailed Dashboard per Call ID

The `Detailed Dashboard per Call ID` displays metrics for each specific `call ID` within a job. Through Grafana's variable filtering, users can select individual `call IDs` or view all `call IDs` associated with a particular job. This dashboard is useful for more in-depth analysis of specific calls within a job, providing detailed insights into their performance.

Note: No images are attached to avoid redundancy, as these visualizations follow a similar structure to the other detailed dashboards.

4.1.4 Detailed Dashboard by Executor ID (Serverless Backend)

This dashboard shows highly granular metrics for child processes of a specific `call ID` within a job. It provides a detailed view of metrics such as CPU and memory usage for each subprocess, making it a valuable tool for advanced users who need deep insights into specific function executions. While not commonly used, some Lithops users have found this dashboard useful in particular cases for troubleshooting or performance tuning.

Note: Similar to the previous dashboards, images are not attached to avoid redundancy.

4.2 Run Lithops Cloud

Run Lithops Cloud is the SaaS version of Lithops that allows users to run custom or predefined notebooks using Lithops. This platform simplifies the execution of serverless tasks while offering robust monitoring features. The focus of this thesis is on the monitoring aspect of the platform.

Within the `Monitoring` page, users can see a list of jobs they've executed. Each job is displayed as a card (Figure 38) that includes the job's name, its associated `executor ID`, and a preview plot that shows the number of `active calls` (i.e., active functions) over time. This preview provides a quick overview of the activity of the job without needing to delve into detailed dashboards.

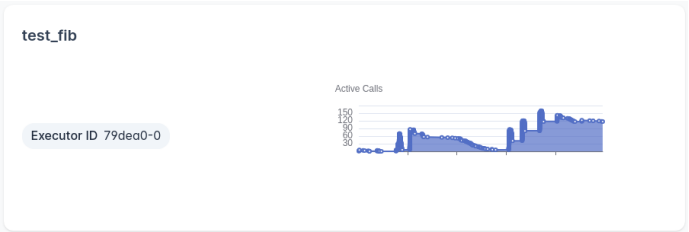

Figure 38: Job Preview in Run Lithops Cloud Monitoring.

To avoid redundancy, no images of the individual metric dashboards will be provided here, as they have already been discussed in the *Profiler Testing* section. However, I include an image (Figure 39) of the `Real-Time Overview`

page, which presents a live summary of the selected job. This page displays key metrics such as average CPU usage, average disk and network usage, average data transfer speed, and average CPU times in real-time.

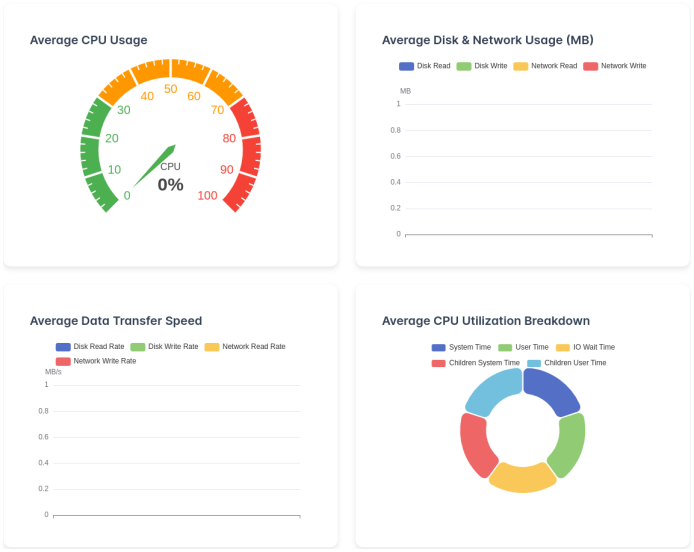

Figure 39: Real-Time Overview in Run Lithops Cloud.

Additional features have been incorporated into the platform, such as the ability to zoom in on all plots, hide specific series to focus on others, switch to a logarithmic scale if necessary, or modify the step of the Prometheus query to adjust the number of data points shown (Figure 40). These enhancements allow for a more flexible and detailed analysis of job performance.

The switch to a logarithmic scale is located at the top-right corner of each plot. Users can zoom in by scrolling or by dragging the temporal range bar below the plot. The step of the query can be adjusted using the slider located at the bottom of the dashboard, allowing for more or fewer data points to be displayed depending on the desired level of detail.

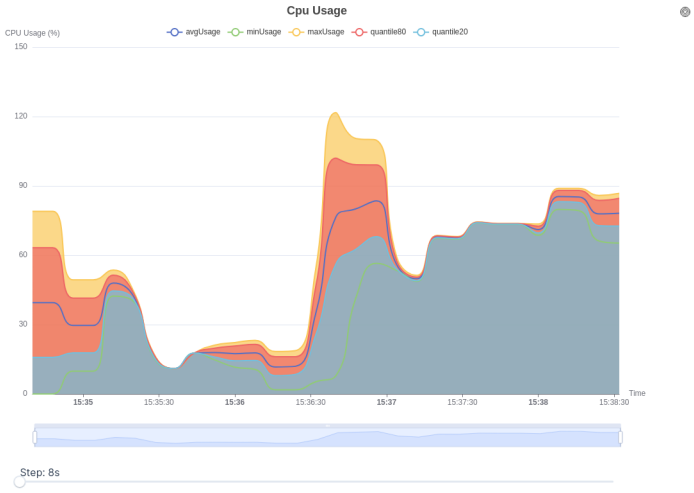

Figure 40: Additional Features for Monitoring in Run Lithops Cloud.

5 Machine Learning Model for Runtime Optimization

5.1 Theoretical Foundations

Predictive Models and Machine Learning: Machine Learning in Resource Optimization: Machine learning (ML) has emerged as a pivotal tool for prediction and optimization in complex systems. It enables the analysis of historical data patterns to forecast future behavior, particularly in dynamic environments such as serverless computing. In these systems, the ability to predict resource usage and execution time is crucial for achieving efficiency. By training models on runtime configurations, input sizes, and job characteristics, ML can provide estimations of the required resources and the optimal number of parallel tasks (or tiles) for minimal execution time.

Application to Monitoring and Scalability: Predictive ML models offer significant advantages in monitoring and resource management. By using real-time data, these models can anticipate resource demand and recommend adjustments to function parallelization, optimizing execution time and reducing latency. This approach is aligned with auto-scaling theories, where resource allocation is dynamically adjusted based on predictive insights, ensuring operational efficiency and improved system responsiveness.

Relevant Studies and Supporting Theories: Several studies have highlighted the critical role of machine learning in optimizing cloud and serverless environments:

[32] provides a foundational framework for understanding the mechanics of machine learning in complex, probabilistic systems. This text is particularly relevant to our goal of using ML models to predict optimal resource allocation based on past execution data.

[13] discusses the co-design of hardware and software in cloud architectures, emphasizing the need for intelligent, data-driven models that can improve system efficiency. This paper supports the notion that ML models can be integrated with serverless platforms to enhance scalability and resource management.

In the context of serverless computing, [44] addresses the use of machine learning to estimate function execution times, a concept that aligns with the predictive modeling approach we aim to implement. Their research demonstrates the feasibility of using ML for performance prediction, which is crucial for optimizing parallelization strategies.

[40] explores the workloads of serverless environments and provides insights into optimizing function execution. This work highlights the potential benefits of machine learning in fine-tuning resource allocation, ensuring that the serverless functions run efficiently without over-provisioning resources.

5.2 Model Application to Runtime Optimization

The proposed machine learning model will be trained using data from multiple job executions in serverless environments. Key features include:

- **Runtime Configuration:** Parameters such as CPU, memory, ephemeral storage, and the number of concurrent threads will be used to capture the configuration of the serverless function.
- **Input Characteristics:** Data related to the number of input files, total input size, and complexity of the tasks will be incorporated to provide a comprehensive understanding of the workload.

5.2.1 Expected Contribution and Innovation

This model aims to fill a gap in the literature by applying machine learning to predict the optimal number of splits or tiles for parallelization in serverless environments, with the goal of minimizing execution time. The number of splits or tiles determines how input data is divided among multiple functions, affecting the overall parallelization. Rather than predicting the ideal resources to allocate, this model focuses on determining the most efficient level of task parallelization based on runtime configuration and input data. While it is currently a standalone predictive tool, this approach provides a more accurate method for optimizing serverless function execution, with the potential for future integration into the Lithops ecosystem.

Supporting Research: [42] discusses the use of machine learning in cloud environments for monitoring and optimization, providing a framework for integrating predictive analytics into cloud workflows. This research supports our approach to predictive monitoring and optimization in serverless platforms.

[26] presents a machine learning model for predictive autoscaling in serverless environments. Their findings show that ML-based models can reduce costs and improve performance by anticipating resource demand, a concept that directly influences our model's design for optimizing function parallelization.

By leveraging machine learning to anticipate and manage resource allocation, this research will contribute to the advancement of serverless computing, setting a new benchmark for operational efficiency, cost management, and system responsiveness.

5.2.2 Future Research Directions

Future research should focus on enhancing the performance of the machine learning model. Additionally, there is potential to explore how data collected from the profiler can enrich the model, improving its predictive accuracy and optimizing execution times even further. The continued integration of real-time data from the profiler and advancements in machine learning algorithms will likely play a pivotal role in evolving serverless computing efficiency [38].

5.3 Technologies Used

The development of the machine learning model for optimizing serverless function execution relies on several key technologies and libraries. These technologies enable data collection, model training, and deployment, facilitating the integration of the predictive model with the Lithops platform.

5.3.1 XGBoost

XGBoost is an optimized gradient boosting algorithm designed to be highly efficient, flexible, and portable. It was chosen for its strong performance in predicting execution times and its ability to handle large-scale datasets with multiple features. The algorithm allows for fine-tuning hyperparameters such as learning rate, max depth, and number of estimators to achieve optimal model performance.

5.3.2 Scikit-learn

Scikit-learn is used extensively for data preprocessing, model evaluation, and feature engineering. The library provides tools for cross-validation, hyperparameter tuning, and performance metrics, making it essential for developing and validating the machine learning model.

5.3.3 Pandas and NumPy

Pandas and NumPy are critical for data manipulation and analysis. These libraries are used to clean, structure, and analyze the historical execution data collected from Lithops jobs. They enable efficient handling of large datasets and facilitate the extraction of meaningful insights through feature engineering.

5.3.4 Matplotlib and Seaborn

Matplotlib and Seaborn are used for data visualization, helping to interpret and analyze the performance of the model. They are employed to plot feature importance, residuals, and evaluation metrics, providing insights into the model's predictive power and accuracy.

5.4 Model Architecture

The architecture of the machine learning model is designed to predict the optimal number of splits (tiles) for parallelizing tasks in a serverless environment. The model processes various runtime and input features to make these predictions, ultimately aiming to minimize execution time.

5.4.1 Data Ingestion

The model ingests historical data from previous Lithops job executions. This data includes key parameters from the runtime configuration, such as:

- Number of input files
- Input size
- Runtime memory
- vCPUs and ephemeral storage
- Worker processes and invoke pool threads
- Execution start and end times

This data is the foundation for training the machine learning model, allowing it to learn how changes in configuration impact the duration of the job.

5.4.2 Feature Engineering

Feature engineering is a critical step in improving the model's predictive accuracy. Several features are created from the raw data to better capture the relationships between the inputs and execution time. Examples include memory per file, storage per input size, and vCPUs per GB. These features enrich the dataset and help the model make more accurate predictions.

5.4.3 Model Training

XGBoost is chosen as the primary model for training. It is highly efficient for tabular data and supports flexible parameter tuning. The model is trained on data labeled with the execution time for each job configuration. By analyzing how the runtime parameters and input data influence execution time, the model learns to predict the number of splits (tiles) that optimize performance.

Additionally, **Random Forest** was tested as an alternative, but **XGBoost** consistently demonstrated better performance in terms of predictive accuracy and computational efficiency, making it the preferred choice for this use case.

5.4.4 Prediction and Validation

Once trained, the model is able to predict the optimal number of tiles for new job configurations. The predictions are validated using real-world job executions, and the results are compared to the predicted values. The profiler tool is used to monitor actual performance, providing feedback that can further refine the model.

5.5 Development Process

The development of the machine learning model was a structured process involving data collection, model training, and evaluation.

5.5.1 Data Collection and Preprocessing

Data was gathered from Lithops executions, capturing various runtime configuration metrics and job characteristics. This historical data forms the backbone of the model's learning process. The data was preprocessed to ensure consistency and formatted into a suitable structure for machine learning, with additional features created through feature engineering.

5.5.2 Feature Augmentation and Synthetic Data

To improve the model's robustness, synthetic data was generated using Gaussian Mixture Models (GMM). This augmented the training data, helping the model perform better in scenarios with limited data availability for certain configurations. Feature engineering also played a crucial role in improving the model's predictive power by creating new features that capture important relationships between runtime configurations and execution times. The following code demonstrates the feature engineering and data augmentation process:

```
def add_features(df):
    """Feature Engineering: Adding new features."""
    df['memory_per_file'] = df['runtime_memory_mb'] / df['num_files']
    df['storage_per_file'] = df['ephemeral_storage_mb'] / df['num_files']
    df['vcpus_per_file'] = df['vcpus'] / df['num_files']
    df['files_per_vcpu'] = df['num_files'] / df['vcpus']
    df['size_per_file'] = df['input_size_gb'] / df['num_files']
    df['memory_per_gb'] = df['runtime_memory_mb'] / df['input_size_gb']
    df['vcpus_per_gb'] = df['vcpus'] / df['input_size_gb']
    df['storage_per_gb'] = df['ephemeral_storage_mb'] / df['input_size_gb']
    df['threads_per_worker'] = df['invoke_pool_threads'] / df['worker_processes']
    df['memory_per_thread'] = df['runtime_memory_mb'] / df['invoke_pool_threads']
    df['vcpus_per_thread'] = df['vcpus'] / df['invoke_pool_threads']
    return df

def augment_data_with_gmm(X, y, num_samples=100):
    """Augment data using Gaussian Mixture Model (GMM) for synthetic data generation."""
    gmm = GaussianMixture(n_components=5, random_state=42)
    gmm.fit(X)

    X_augmented = gmm.sample(num_samples)[0]

    # Augment y to match the number of samples generated by GMM
    y_augmented = np.tile(y, num_samples // len(y) + 1)[:num_samples]

    return X_augmented, y_augmented
```

Listing 2: Feature Engineering and Data Augmentation with GMM

This code first creates new features to better capture relationships between input parameters and execution performance. Next, it uses a Gaussian Mixture Model (GMM) to generate synthetic data, augmenting the dataset and improving the model's robustness in scenarios with limited data.

The feature importance plot (Figure 41) shows that the number of files and

the number of tiles are the most critical features in predicting execution time. The number of tiles directly affects the degree of parallelization, which has a significant impact on the overall job performance. Additionally, runtime memory and input size also contribute to performance, although to a lesser extent. Meanwhile, vCPUs have a smaller influence, suggesting that the distribution of data (through tiles) and memory allocation plays a more decisive role in optimizing execution duration.

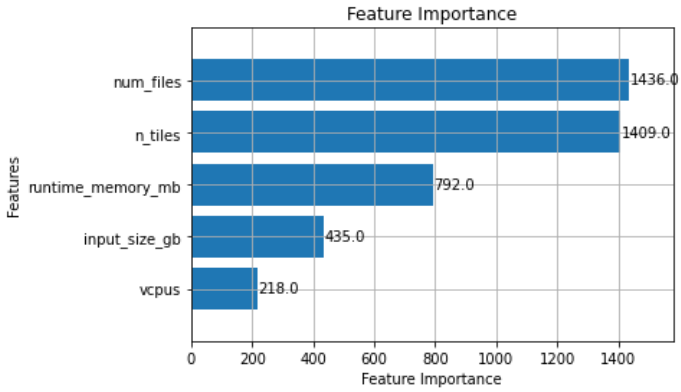

Figure 41: Feature Importance in Predicting Execution Time

5.5.3 Model Selection and Training

Several models were considered, with **XGBoost** being selected due to its performance with large datasets and high-dimensional feature spaces. Hyperparameter tuning was conducted using `GridSearchCV`, optimizing parameters such as learning rate, max depth, and number of estimators to improve accuracy. The following code snippet demonstrates the process of tuning the hyperparameters:

```
def randomized_search_xgboost(X, y):
    """Perform Randomized Search with Cross-Validation
        to find the best XGBoost parameters."""
    # Define the XGBoost model
    xgb_model = xgb.XGBRegressor(objective='reg:
        squarederror')

    # Define the parameter grid for tuning
    param_dist = {
        'learning_rate': [0.01, 0.05, 0.1],
        'max_depth': [3, 5, 7, 10],
        'n_estimators': [100, 200, 500],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'gamma': [0, 0.1, 0.2],
        'reg_alpha': [0, 0.1, 1],
        'reg_lambda': [0, 0.1, 1]
    }

    # Perform the grid search
    with parallel_backend('threading'):
        grid_search = GridSearchCV(
            estimator=xgb_model,
            param_distributions=param_dist,
            n_iter=20,  # Number of combinations to
                test
            cv=5,
            scoring='neg_mean_squared_error',
            verbose=2,
            n_jobs=-1
        )

        # Fit the model
        grid_search.fit(X, y)

        # Output the best parameters and best score
        print("Best␣Parameters:", grid_search.
            best_params_)
        print("Best␣Score:", np.sqrt(-grid_search.
            best_score_))

    return grid_search.best_estimator_
```

Listing 3: Hyperparameter Tuning using GridSearchCV with XGBoost

This code performs hyperparameter tuning by searching over a range of parameters for the best combination, ultimately selecting the most optimal configuration for the XGBoost model based on cross-validation results.

5.5.4 Model Evaluation and Optimization

The model was evaluated using negative mean squared error (`neg_mean_squared_error`) on test datasets. Outliers were handled by removing extreme values using interquartile range (IQR). Cross-validation was employed to ensure that the model generalized well across different scenarios, avoiding overfitting.

```
Q1 = df_metrics['duration'].quantile(0.25)
Q3 = df_metrics['duration'].quantile(0.75)
IQR = Q3 - Q1

# Define bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Remove outliers
df_metrics = df_metrics[(df_metrics['duration'] >=
    lower_bound) & (df_metrics['duration'] <=
    upper_bound)]
```

Listing 4: Outlier Removal Using Interquartile Range

As shown in Figures 42 and 43, the original dataset contained outliers that significantly deviated from the majority of data points. After executing the outlier removal code using the interquartile range (IQR) method, the outliers were successfully removed, resulting in a cleaner dataset with more consistent values.

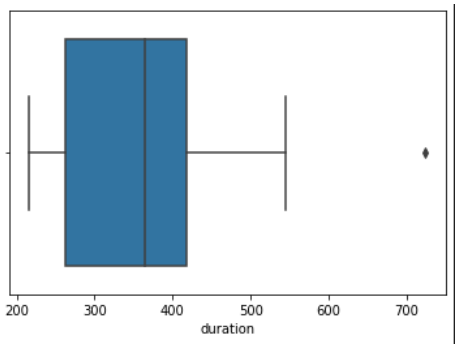

Figure 42: Dataset with Outliers

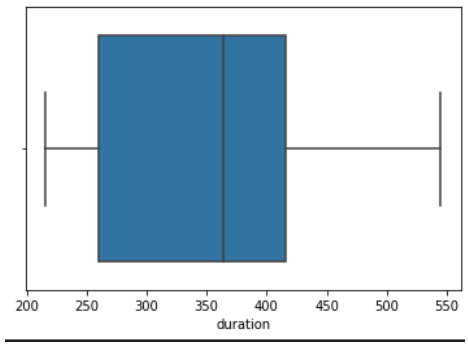

Figure 43: Dataset after Outlier Removal

5.6 Results

Data Source and Experimental Setup The data used for these experiments originates from the executions of the `Water Consumption Pipeline`, a serverless application designed to estimate water consumption based on geospatial data. This pipeline was run multiple times under different configurations, generating a range of execution metrics such as input size, number of tiles, memory usage, and execution duration.

These execution logs formed the foundation for training the machine learning model, allowing it to learn the relationship between runtime configuration, input data, and execution time. By analyzing how the number of splits (or tiles) affects the parallelization of tasks and the overall job duration, the model aims to predict the optimal configuration for future executions.

The model was trained and tested using the historical execution data of this specific pipeline, and the results presented in the following sections demonstrate the model's ability to predict execution time based on various configurations of the `Water Consumption Pipeline`.

Model Validation and Residual Analysis The residual plot (Figure 44) illustrates the difference between the predicted and actual values after training the model. The points are scattered around the horizontal axis, representing the zero error line. Although most points are close to this axis, a few outliers indicate that the model does not perfectly predict execution times for all jobs.

The model's performance was evaluated using the mean absolute error (MAE), which measures the average magnitude of errors between the predicted and actual values. In this case, the MAE was 31.06 seconds, indicating that, on average, the predicted execution times deviate by 31.06 seconds from the actual times. While this error is within a reasonable range, it suggests that there is still room for improvement in the model.

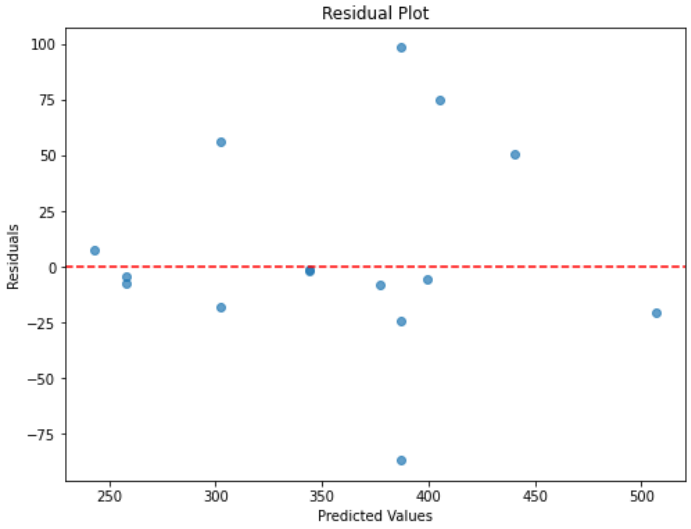

Figure 44: Residual Plot.

The training data, denoted as `X_train`, consisted of historical execution information, including parameters such as input size, number of tiles, memory usage, and runtime configurations. The model was trained using these features and the corresponding target variable: execution duration. The test data, labeled as `X_test`, was used to evaluate the model's performance.

For this test case, the following configuration was used as the `X_test` input:

```
test_data = pd.DataFrame({
    'num_files': [15],
    'n_tiles': [3],
    'input_size_gb': [1.0],
    'runtime_memory_mb': [2850],
    'ephemeral_storage_mb': [512],
    'worker_processes': [1],
    'invoke_pool_threads': [64],
    'vcpus': [1.61],
})
```

To predict the optimal number of tiles, the function `predict_optimal_tiles` was called, passing a set of possible tile configurations: `[9, 16, 25]`. This function predicts the execution duration for each configuration and selects the one with the shortest predicted duration. The results were as follows:

- **9 tiles**: 344.31 seconds
- **16 tiles**: 405.11 seconds
- **25 tiles**: 440.28 seconds

Based on these predictions, the optimal number of tiles for this job configuration was **9**.

After running the job with the configuration in `test_data`, the actual duration observed in the Gantt chart (Figure 45) was approximately 368 seconds. The difference between the predicted duration for 9 tiles (344.31 seconds) and the observed duration (368 seconds) is:

$$368\,\text{seconds} - 344.31\,\text{seconds} = 23.69\,\text{seconds}$$

This difference of 23.69 seconds is within the expected error margin, as the model's MAE was calculated to be 31.06 seconds. This suggests that the prediction was reasonably accurate, given the inherent variability in execution times.

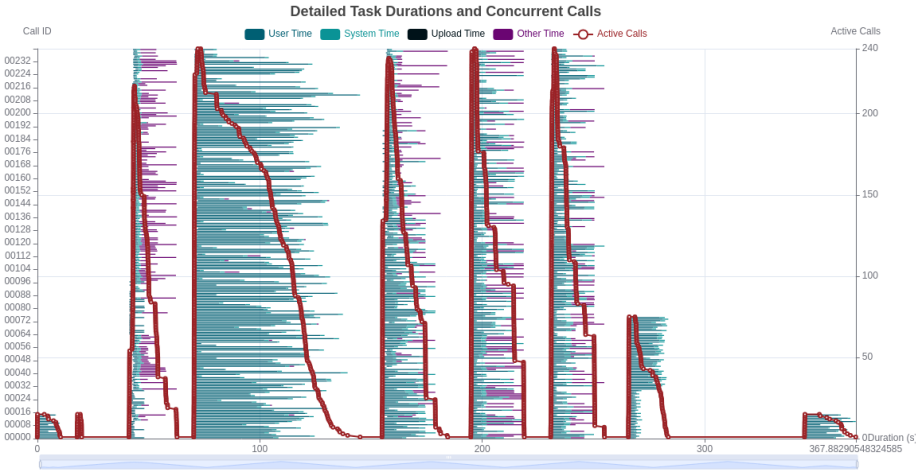

Figure 45: Gantt chart showing the observed duration of approximately 368 seconds for the configuration with 9 tiles

Explanation of the `predict_optimal_tiles` Function The function `predict_optimal_tiles` takes a trained model, a data scaler, test data, and a list of possible tile configurations. It calculates the predicted execution duration for each configuration and returns the optimal number of tiles that minimizes the duration. The process involves:

- Modifying the test data by adjusting the `n_tiles` value for each possible configuration.
- Adding additional features through the `add_features` function.
- Scaling the test data using the provided scaler to ensure consistency with the model's training data.
- Predicting the execution duration using the trained model.

- Comparing the predicted durations and selecting the configuration with the shortest execution time.

The following code snippet demonstrates the function:

```
def predict_optimal_tiles(model, scaler, test_data, possible_tiles, feature_columns):
    """Predicts the optimal number of tiles that minimizes the duration for RandomForest."""

    optimal_tiles = None
    min_duration = float('inf')

    for tiles in possible_tiles:
        # Add the value of n_tiles to the DataFrame
        test_data['n_tiles'] = tiles

        # Add additional features using the add_features function
        test_data_with_features = add_features(test_data.copy())

        # Ensure the columns are in the same order as used during training
        X = test_data_with_features[feature_columns]

        # Scale the features
        X_scaled = scaler.transform(X)

        predicted_duration = model.predict(X_scaled).mean()

        print(f"Predicted duration for {tiles} tiles: {predicted_duration}")

        # Check if this prediction has the minimum duration
        if predicted_duration < min_duration:
            min_duration = predicted_duration
            optimal_tiles = tiles

    return optimal_tiles
```

Listing 5: Predicting Optimal Tiles for Minimizing Execution Time

5.7 Conclusions

The development of a machine learning model to predict the optimal number of tiles for parallelization in serverless environments has shown promising results,

with the model achieving a mean absolute error (MAE) of 31.06 seconds. While the predictions are valuable, further improvement could be achieved with a larger dataset.

Due to the cost of running additional controlled executions, the next step is to leverage real-world data from `Run Lithops Cloud` users. This would allow the model to continuously refine its accuracy across different workloads and configurations.

The integration of the profiler and monitoring architecture has helped validate the model's predictions, ensuring consistency with real runtime data. Future work could involve a deeper analysis of other metrics, such as CPU and memory usage, to further enhance performance optimization. Ultimately, the goal is to integrate the model into the Lithops ecosystem, providing users with an efficient tool for optimizing serverless executions.

6 Final Reflections and Future Directions

This thesis presented the development and integration of a comprehensive monitoring and optimization system for serverless environments, focusing on `Lithops` and its platform, *Run Lithops Cloud*. At the heart of this work lies the *Profiler*, a tool designed to gather real-time metrics on resource usage across distributed jobs, helping to identify performance bottlenecks, optimize resource allocation, and improve parallel execution.

As discussed throughout the thesis, two versions of the *Profiler* were created—one for the open-source version of `Lithops`, leveraging `Prometheus` and `Grafana`, and another for *Run Lithops Cloud*, which integrates with `Managed Prometheus` and provides custom dashboards using `Apache ECharts`. These dashboards, whether through `Grafana` or *Run Lithops Cloud*, enable users to monitor their workloads in real-time and gain detailed insights into system performance.

Additionally, a machine learning model was designed to predict the optimal number of tiles (or splits) for efficient parallel execution in serverless environments. This model proved capable of providing reasonably accurate predictions, with a mean absolute error of 31.06 seconds, offering valuable guidance for optimizing workloads. However, there is still potential for improving the precision and adaptability of the model.

6.1 Key Contributions

The key contributions of this thesis are:

- **Development of the Profiler:** The `Profiler` has proven to be a robust monitoring solution for tracking CPU, memory, disk, and network usage across distributed jobs. Its architecture allows scalability, but overhead issues were identified when many functions ran in parallel, even with `Managed Prometheus`.

- **Customizable Dashboards:**
  - The open-source `Grafana` dashboards offer flexibility to users, enabling detailed tracking of jobs and resource usage.
  - In *Run Lithops Cloud*, `Apache ECharts` dashboards simplify monitoring, providing real-time visualizations within the platform itself, without requiring additional configuration.
- **Machine Learning for Runtime Optimization:** The machine learning model, developed for predicting the optimal number of tiles for job parallelization, demonstrated potential for reducing execution times by optimizing the task distribution and resource usage. However, further refinement of this model is needed to improve its accuracy and performance in different scenarios.
- **Community Feedback:** The `Profiler`, alongside the `Grafana` dashboards and the *Run Lithops Cloud* platform, was presented to the members of *CloudLab*. The feedback from the community was positive, with recognition of the system's practical value in improving serverless workload management. However, it was acknowledged that there is still room for improvement, especially in terms of scaling and further integration of the model with the `Profiler`.

6.2 Future Work

Several avenues for future research and improvement have been identified:

- **Enhanced Integration Between Profiler and Machine Learning Model:** There is a significant opportunity to improve the integration between the `Profiler` and the machine learning model, allowing for real-time optimization based on live resource monitoring. This could lead to dynamic adjustments in workload configurations, improving performance during execution.
- **Improving Model Precision:** The machine learning model can be further improved by incorporating additional data from real-world job executions in *Run Lithops Cloud*. With more data, the model can generalize better across different workloads, leading to more accurate predictions.
- **Addressing Overhead in Large-Scale Parallelism:** The overhead encountered during high levels of parallel execution remains a challenge. Future research should focus on refining how metrics are collected and processed, especially in cases with hundreds of concurrent functions, to minimize the impact on performance.
- **Extending Metrics for Deeper Optimization:** Incorporating additional metrics—such as CPU time per thread, disk I/O, and network latency—could enhance the depth of optimization. This would allow the

`Profiler` to provide even more granular insights into performance, enabling finer control over job configurations.

6.3 Final Thoughts

As I reach the conclusion of this thesis, it becomes clear that significant progress has been made in narrowing the gap between real-time monitoring and machine learning-driven optimization in `Lithops`. The development of the `Profiler`, along with the flexible dashboards and machine learning model, has opened the door to more intelligent and efficient management of distributed workloads. The positive feedback from the *CloudLab* members confirms the value of this work, but I know there's still room for improvement.

Challenges remain—addressing overhead issues in large-scale parallelism, improving the accuracy of the machine learning model, and refining the integration between monitoring and optimization. These are areas that offer great potential to further advance serverless computing.

Looking ahead, I feel a growing sense of excitement. The groundwork has been laid, the tools have been built, and feedback has been taken on board. With each step, we move closer to creating a more intelligent, adaptable, and efficient system. The potential is immense, and I'm eager to continue this journey. There's still much to explore, and the future holds endless opportunities.

7 References

[1] Daniel Aguilera, Pedro Carreira, Enric Borràs, and Boris Alomar. Lithops: A multipurpose python library for serverless computing. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 155–156, 2021.

[2] Kubernetes Authors. *Kubernetes Documentation: Orchestration for Automating Deployment, Scaling, and Management*, 2023. Accessed: 2024-09-04.

[3] Prometheus Authors. *Prometheus: Monitoring System and Time Series Database Documentation*, 2023. Accessed: 2024-09-04.

[4] Amazon Web Services (AWS). *AWS Lambda Documentation*, 2023. Accessed: 2024-09-04.

[5] Amazon Web Services (AWS). *AWS Managed Service for Prometheus*, 2023. Accessed: 2024-09-04.

[6] Ioana Baldini, Paulo Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, ..., and Marco Pistoia. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[7] Salman A. Baset. Cloud slas: Present and future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[8] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016. Accessed: 2024-09-04.

[9] Y. Chen, X. Zhao, and S. Liu. Evaluating the performance of aws x-ray in serverless architectures. *Journal of Cloud Computing*, 7(2):145–158, 2018.

[10] Lithops Community. *Lithops - Cloud-agnostic Python Serverless Programming Platform*, 2023. Accessed: 2024-09-04.

[11] Julien Danjou and contributors. *Tenacity - Retrying Library for Python Documentation*, 2023. Accessed: 2024-09-04.

[12] Isaac de la Higuera, Javier Serrano, and Boris Alomar. Lithops: Serverless mapreduce for large-scale bioinformatics applications. *Bioinformatics*, 38(7):2131–2137, 2022.

[13] Jeffrey Dean, David A. Patterson, and Randy H. Katz. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, and beyond. *IEEE Micro*, 38(2):21–29, 2018.

[14] Markus Dieter et al. Learning to auto-scale serverless functions with reinforcement learning. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 199–210. ACM, 2020.

[15] John D. Hunter et al. *Matplotlib: Visualization with Python*, 2023. Accessed: 2024-09-04.

[16] Travis Oliphant et al. *NumPy*, 2023. Accessed: 2024-09-04.

[17] M. Garcia, T. Lee, and H. Kim. Complexities of multi-cloud monitoring: A case study with new relic and stackdriver. In *Proceedings of the International Conference on Cloud Computing*, pages 101–110. IEEE, 2019.

[18] Grafana Labs. *Grafana Documentation: Adjusting Data Points per Minute*, 2022.

[19] Animesh Gupta, Amit Kumar, and Pedro Carreira. Challenges in scaling serverless computing. *Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E)*, pages 98–109, 2020.

[20] J. Hayes and A. Komar. Scalability challenges in monitoring serverless architectures. *IEEE Transactions on Cloud Computing*, 8(4):450–462, 2020.

[21] Jian Huang et al. Serverless computing: Machine learning optimizations and challenges. *ACM Computing Surveys*, 54(5):1–36, 2021.

[22] iAmJK44. Serverless pipelines: Lithops-metaspace, 2024. Accessed: 2024-09-04.

[23] R. Kumar and S. Patel. Adapting prometheus and grafana for serverless computing. *ACM Computing Surveys*, 53(3):212–229, 2021.

[24] Grafana Labs. *Grafana Documentation*, 2023. Accessed: 2024-09-04.

[25] Lithops Project. *Lithops Documentation: Metrics and Monitoring*, 2022.

[26] Wei Liu et al. Predictive autoscaling for serverless functions with machine learning. *IEEE Transactions on Cloud Computing*, 9(1):2–15, 2020.

[27] Y. Liu and W. Zhang. Real-time data processing in serverless environments: A review. *Journal of Internet Services and Applications*, 12(1):47–65, 2022.

[28] Gavin McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–10. IEEE, 2017.

[29] Wes McKinney. *Pandas - Python Data Analysis Library*, 2023. Accessed: 2024-09-04.

[30] Brian Morris and Michael Anderson. Multicloud architecture: The future of cloud computing. *Journal of Cloud Computing*, 8(1):22–35, 2019.

[31] J. Morrison. The need for cloud-agnostic monitoring tools in multi-cloud environments. In *Proceedings of the International Workshop on Cloud Computing*, pages 67–74. ACM, 2021.

[32] Kevin P. Murphy. *Machine learning: A probabilistic perspective*. MIT press, 2012.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. *scikit-learn: Machine Learning in Python*, 2011. Accessed: 2024-09-04.

[34] Prometheus.io. *Prometheus: Monitoring System and Time Series Database*, 2021. Accessed: 2021-09-15.

[35] Giampaolo Rodola. *psutil - Python cross-platform library for process and system monitoring*, 2023. Accessed: 2024-09-04.

[36] Giampaolo Rodola and contributors. *psutil GitHub Issues: Known Limitations and Troubleshooting*, 2023. Accessed: 2024-09-04.

[37] Maria Rodriguez, Gerardo Vega, and Rich Wolski. Serverless computing for data-intensive applications. *ACM Transactions on Internet Technology (TOIT)*, 21(1):1–22, 2021.

[38] Pablo Rodriguez et al. Scaling serverless functions: Insights from real-world workloads. *Journal of Cloud Computing*, 9:23–37, 2020.

[39] Josep Sampé, Aitor Arjona, Sergi Domingo, Arnau Gabriel, Sara Lanuza, Daniel Alejandro Coll, Sergi Alberich, and Matrix Foundation Former authors: Answare Tech. Cloudbutton geospatial use case: Water consumption. `https://github.com/cloudbutton/geospatial-usecase/tree/main/water-consumption`, 2023.

[40] Maryam Shahrad, Rad Behnam, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858. ACM, 2019.

[41] A. Smith et al. An evaluation of monitoring tools in serverless computing. *Journal of Systems and Software*, 159:110–123, 2020.

[42] Michael Volz and Chris Birkner. *Monitoring cloud-native applications: Prometheus and beyond*. O'Reilly Media, 2020.

[43] P. Wilson and L. Zhao. Challenges in distributed tracing for serverless functions. *IEEE Cloud Computing*, 7(3):32–40, 2020.

[44] Rich Wolski et al. Estimating execution time for serverless functions with machine learning. *Journal of Cloud Computing*, 8:12–24, 2019.

[45] Neeraja Yadwadkar, Sriram Hariharan, and Peter Bailis. Fast and low-cost serverless computing using Spot Instances. In *Proceedings of the 2017 ACM Symposium on Cloud Computing*, pages 123–133, 2017.

8 Appendix: Lithops Profiler Configuration Tutorial

A Lithops Configuration

A.1 Profiler Timeout

The `profiler_timeout` setting determines the frequency (in seconds) at which the Lithops Profiler collects performance metrics. The default value is set to 1 second, offering a fine-grained view of the system's performance. Adjusting this value can help balance the level of detail in the performance data collected with the overhead introduced by the monitoring process.

```
lithops:
    profiler_timeout: 1 # Default value
```

A.2 Prometheus Configuration

For more details on configuring Prometheus for use with Lithops, please refer to the dedicated section in the [25].

A.2.1 Key Configuration Options

When integrating Prometheus with Lithops for monitoring, it's crucial to adjust the default configuration to ensure no data is lost. By default, Prometheus is configured to scrape metrics every 15 seconds, which might not be frequent enough to capture all metrics sent by Lithops, especially if metrics are emitted every second.

- **scrape_interval**: Specifies how often Prometheus should scrape metrics from monitored targets. To capture all metrics from Lithops, you should set this value to match or exceed the frequency at which metrics are generated.
- **scrape_timeout**: Specifies the maximum time Prometheus waits for a scrape request to complete. Setting this appropriately ensures that Prometheus does not time out while scraping metrics if the response is slightly delayed.

For more information, refer to the official documentation [34].

Additionally, to learn how to adjust data points per minute in Grafana to reduce costs, visit the official Grafana documentation [18].

A.3 Using a Remote Backend with Prometheus

In scenarios where Prometheus and Lithops are operating in different environments, such as Lithops running on AWS Lambda and Prometheus on an EC2

instance, a remote backend setup becomes necessary. This section outlines the steps to configure Prometheus to scrape metrics from a remote Pushgateway, allowing for aggregation and monitoring of metrics from serverless functions or other distributed systems.

1. **Deploying Prometheus on EC2:** Start by installing Prometheus on your EC2 instance. This can be achieved through package managers like `apt-get` for Debian-based systems:

   ```
   sudo apt-get update
   sudo apt-get install prometheus
   ```

   Ensure that Prometheus is configured to start on boot and is running properly.

2. **Setting up Pushgateway with Docker:** Unlike the local setup where Pushgateway might be installed via `apt-get`, in a cloud environment like EC2, deploying Pushgateway with Docker is recommended.

   ```
   docker run -d -p 9091:9091 prom/pushgateway
   ```

   This command will start Pushgateway and expose it on port `9091`, making it accessible for receiving metrics.

3. **Security Group Configuration:** Adjust the EC2 instance's security group settings to allow inbound traffic on ports `9090` (Prometheus) and `9091` (Pushgateway) to ensure external accessibility.

4. **Prometheus Configuration:** Modify the `prometheus.yml` configuration file to include the Pushgateway as a target for scraping:

   ```
   scrape_configs:
     - job_name: 'pushgateway'
       static_configs:
         - targets: ['localhost:9091']
   ```

5. **Lithops Configuration for Remote Metrics:** In your `lithops_config` file, specify the remote Pushgateway endpoint:

   ```
   prometheus:
     pushgateway_url: 'http://<your-ec2-instance-public-ip>:9091'
   ```

   Replace `<your-ec2-instance-public-ip>` with the actual public IP address of your EC2 instance.

By following these steps, you can successfully set up Prometheus to monitor Lithops metrics using a remote backend, ensuring comprehensive visibility across distributed computing environments.

A.4 Using a Local Backend with Prometheus

Setting up a local backend for Prometheus to monitor Lithops is similar to the remote configuration. The main difference lies in the deployment environment, but the overall process remains consistent.

B AWS Managed Prometheus Integration with Lithops

AWS Managed Prometheus simplifies the setup by providing a fully managed Prometheus service, eliminating the need for manual infrastructure management. To integrate AWS Managed Prometheus with Lithops, follow these steps:

B.1 Creating an AWS Managed Prometheus Workspace

1. In the AWS Management Console, navigate to **Amazon Managed Service for Prometheus**.
2. Create a new Prometheus workspace, and note the **workspace endpoint** provided after creation.

B.2 Configuring Lithops to Use AWS Managed Prometheus

In the `lithops_config` file, specify the endpoint of the AWS Managed Prometheus service as follows:

```
prometheus:
    pushgateway_url: 'https://<your-prometheus-workspace-endpoint>'
```

Replace `<your-prometheus-workspace-endpoint>` with the actual endpoint of your AWS Managed Prometheus workspace.

By leveraging AWS Managed Prometheus, you benefit from a fully managed monitoring service, ensuring scalability, security, and reliability for your distributed systems. The only configuration change required in Lithops is specifying the managed service's endpoint, similar to previous local and remote setups.

Note

When using self-managed Prometheus, scalability issues may arise if a large number of functions run in parallel. Prometheus may struggle to handle the volume of metrics generated by many concurrent functions, potentially leading to performance bottlenecks. It is advisable to monitor Prometheus's performance, adjust resource allocations, or consider using a managed solution like AWS Managed Prometheus for large-scale deployments.