Design and Implementation of a Telemetry Platform for High-Performance Computing Environments

Ole Christian Weidner

Doctor of Philosophy

Artificial Intelligence Applications Institute

School of Informatics

University of Edinburgh

2021

Abstract

A new generation of high-performance and distributed computing applications and services rely on adaptive and dynamic architectures and execution strategies to run efficiently, resiliently, and at scale in today's HPC environments. These architectures require insights into their execution behaviour and the state of their execution environment at various levels of detail, in order to make context-aware decisions. HPC telemetry provides this information. It describes the continuous stream of time series and event data that is generated on HPC systems by the hardware, operating systems, services, runtime systems, and applications. Current HPC ecosystems do not provide the conceptual models, infrastructure, and interfaces to collect, store, analyse, and integrate telemetry in a structured and efficient way. Consequently, applications and services largely depend on one-off solutions and custom-built technologies to achieve these goals; introducing significant development overheads that inhibit portability and mobility. To facilitate a broader mix of applications, more efficient application development, and swift adoption of adaptive architectures in production, a comprehensive framework for telemetry management and analysis must be provided as part of future HPC ecosystem designs.

This thesis provides the blueprint for such a framework: it proposes a new approach to telemetry management in HPC: the *Telemetry Platform* concept. Departing from the observation that telemetry data and the corresponding analysis, and integration patterns on modern multi-tenant HPC systems have a lot of similarities to the patterns observed in large-scale data analytics or "Big Data" platforms, the telemetry platform concept takes the data platform paradigm and architectural approach and applies them to HPC telemetry. The result is the blueprint for a system that provides services for storing, searching, analysing, and integrating telemetry data in HPC applications and other HPC system services. It allows users to create and share telemetry data-driven insights using everything from simple time-series analysis to complex statistical and machine learning models while at the same time hiding many of the inherent complexities of data management such as data transport, clean-up, storage, cataloguing, access management, and providing appropriate and scalable analytics and integration capabilities.

The main contributions of this research are (1) the application of the data platform concept to HPC telemetry data management and usage; (2) a graph-based, time-variant telemetry data model that captures structures and properties of platform and applications and in which telemetry data can be organized; (3) an architecture blueprint and prototype of a concrete implementation and integration architecture of the telemetry platform; and (4) a proposal for decoupled HPC application architectures, separating telemetry data management, and feedback-control-loop logic from the core application code. First experimental results with the prototype implementation suggest that the telemetry platform paradigm can reduce overhead and redundancy in the development of telemetry-based application architectures, and lower the barrier for HPC systems research and the provisioning of new, innovative HPC system services.

Lay Summary

High-Performance Computing (HPC) describes the theory and practice of building and operating computer systems and programs that are thousands of times larger and faster than personal computers. HPC helps to answer many important science questions across many disciplines, such as physics, biology, climate sciences, and medicine.

HPC systems and programs can easily fail due to their extreme size and complexity. Failures, such as HPC programs crashing without producing any results, are generally expensive because of the computing resources, and, ultimately the electrical energy they waste. To avoid failure, HPC software developers try to build so-called resiliency mechanisms into their programs that allow them to detect potential failures ahead of time and either mitigate them or exit the program gracefully.

Many of these mechanisms rely on information on the past and current state of the HPC system, its hardware, and the programs that run on it. This information is called *HPC telemetry*. Software developers face two main problems when working with HPC telemetry: (1) telemetry is difficult and time-consuming to access as the majority of HPC systems do not expose it directly to their users, and (2) telemetry is difficult to process and analyse at the scale of modern HPC systems and programs. As a consequence, many HPC programs, especially the ones with limited development resources available, cannot invest into resiliency mechanisms.

In this work, we propose a system that can help software developers and users to integrate HPC telemetry more easily and efficiently in their applications. We call this system a *Telemetry Platform*. Telemetry platforms can collect and store the telemetry data that is generated during the operation of an HPC system in a large database. The database is organized in a way that the data can easily be accessed, understood and analysed by all users of the system. Telemetry platforms are different from other, similar systems in that they also provide the building blocks that allow HPC application developers to easily integrate telemetry with their program logic. These integrations, also called feedback loops, are important to realize automatic resiliency mechanisms at a low level of complexity and effort.

To verify the telemetry platform idea, we build and evaluate a prototype called SEASTAR and show how it can be used to efficiently collect and process telemetry data and to build feedback loops based on the analysis of this data. This helps us understand how we can build better, more user-friendly and resource-efficient HPC environments in the future.

Acknowledgements

First and foremost I am extremely grateful to my supervisors, Professor Malcolm Atkinson at the University of Edinburgh, Professor Rosa Filgueira at Heriot-Watt University, and Professor Adam Barker at the University of St. Andrews for their direction, invaluable advice, and most of all patience during my ten years of PhD study. In particular, I would like to thank Professor Atkinson for his tireless encouragement and empathy that kept me going through the many challenging phases a part-time PhD brings with it. I would also like to thank my former research groups at Louisiana State University and Rutgers University, New Jersey, Professor Shantenu Jha, Professor Hartmut Kaiser, Andre Merzky, and Professor Matteo Turilli. It was those formative years I spent with them as a distributed systems researcher and software engineer that has laid the groundwork for this work. I would like to thank Amazon Webservices Inc. for their generous contribution through their AWS in Education Research Grant. Without it, the practical realization of this work would not have been possible. Similarly, I would like to thank the Edinburgh Parallel Computing Centre, Professor Mark Parsons in particular, for granting me access to their HPC clusters and workload manager statistics database to carry out my preliminary investigations. I would like to thank Professor Dr H. Ibrahim Uzun, who kindly granted me access to a vacant office at the Agricultural Faculty at Akdeniz Üniversitesi, Antalya, Turkey, where I wrote the bulk part of this thesis in Spring 2020. I also have to thank Yeter Yurdakan for the constant supply of tea and pastries during that time. Finally, I must express my infinite gratitude to my partner Nil. Without her tremendous understanding, encouragement support, and sparring, it would have been impossible for me to complete this research.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ole Christian Weidner*)

Dedicated to Nil and Leon

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

"This is a much broader definition of monitoring than many take. My definition could be summed up as the data and metadata that show the state of our entire facility, at any instant in time, at any level of detail. And I want a pony too."

— William (Bill) Allcock, Argonne National Laboratory

We define telemetry as the continuous stream of operational data that is generated on High-Performance Computing (HPC) platforms by the hardware, operating systems, services, runtime systems, and applications. Temperature and power consumption readings from hardware sensors, job and process metrics, network and disk I/O metrics, and MPI message size metrics are all examples of telemetry data generated on an HPC system. Telemetry provides insights into the current and past states of an HPC system and the applications running on it. It also provides the foundation to make predictions about their future behaviour.

With growing system size, the departure from traditional, tightly coupled, homogeneous HPC applications and the advent of a new generation of system services and applications with more heterogeneous and dynamic properties, such as dynamic and autonomic computing approaches, the role of telemetry is changing. On today's large-scale HPC systems, telemetry is not only crucial for the efficient operation of the platform, but is becoming increasingly relevant for system services such as schedulers, data-transfer services, and end-user applications to monitor and optimize their operation and to detect and diagnose system failures and trigger remedial actions.

Despite its importance for today's applications, telemetry has only been playing a secondary role in the design and architecture of contemporary HPC ecosystems. The concepts and tools to collect, store and analyse telemetry data are heavily fragmented and segregated into the system operations, parallel frameworks, and end-user application domains with little to no cross-fertilization. There is no model, system, or programming interface that would allow telemetry consumers across these domains to tap into the rich stream of information generated on these systems.

This lack of a common strategy and approach to telemetry creates significant development overheads and redundancy, and a wild-growth of localized solutions that hamper application portability and mobility. This, in turn, impedes the development and adoption of adaptive optimization and resilience architectures, a critical capability, especially for exascale systems and applications. The lack of a common approach also effectively prevents applications and experiments from "learning from each other" through a shared repository of historic data, from which for example, tuning parameters from previous application runs could be derived. It also prevents platform operations from gaining an in-depth understanding of the application landscape, which would be incredibly valuable to optimize platform parameters and strategically plan future system extensions and modifications. Lastly, the lack of systematic support for telemetry data largely prevents the practical application of novel machine learning and artificial intelligence-based approaches for application optimization, failure prediction and other advanced resiliency techniques.

To tackle these issues, telemetry must play a more central role in HPC ecosystem capability designs. Telemetry must become a first-order platform service, with the same status as job management, data storage, or backup services. This research sets out to fill this gap by defining a new approach to working with telemetry on HPC systems: the *Telemetry Platform*. We start with the observation that telemetry data and the corresponding analysis, and integration patterns on modern multi-tenant HPC systems look very similar to the patterns and workflows observed in large-scale data-science and internet-of-Things (IoT) applications. This work takes the *data platform* architecture that supports these patterns and workflows and applies it to HPC telemetry. The result is a telemetry platform that provides services for storing, searching, understanding, analysing, and integrating telemetry data in HPC applications and system services.

This work lays out a blueprint for the next evolutionary step in telemetry management and usage. It builds upon and extends the concepts and ideas of existing HPC monitoring solutions that have been traditionally focusing on systems operations use cases and have been slow to incorporate the requirements and use cases of a growing and diverse application landscape and the new and unique challenges of upcoming exascale HPC applications. It is important to understand that this approach does not aim to replace existing monitoring and telemetry solutions, but rather builds on them and integrates them into a framework that provides telemetry management and usage functionality to application users, developers, systems researchers, and platform operators alike.

1.1 Motivation and Research Question

In order to cope with the growing size, and complexity of HPC systems and applications, new techniques for optimization and resilience are being developed at a steady pace. Many, if not all of these techniques require access to information about the environment in which they execute in order to make decisions about future application configurations and runtime trajectories. While great progress has been made in the areas of adaptive, autonomous, and resilient applications, adoption of these patterns and techniques has remained difficult due to several challenges with telemetry management capabilities on HPC systems. These may manifest as:

(i) **Availability:** telemetry is collected by monitoring systems, schedulers and other system services to support the HPC operations teams, but the data is not made available to a broader audience.

(ii) **Accessibility:** telemetry is not available in machine-readable formats or the formats diverge between different platforms and services. The same holds for the interfaces exposing telemetry data which can range from Application programming interface (API) endpoints to flat-file downloads from a website.

(iii) **Integration:** for many use cases, telemetry or insights derived from telemetry analysis need to be integrated back into applications and services as input for optimization and resilience functions. No common approaches and patterns exist for this integration.

(iv) **Structure and Semantics:** telemetry is often difficult if not impossible to interpret without having an implicit understanding of the semantic context and the structure of the HPC system and application that was generating it. Little effort has been put into defining a framework for this.

(v) **Processing and Analysis:** telemetry data volumes can quickly become very large at scale which requires advanced technology and infrastructure for processing and analysis. This can often not be done within the scope of an HPC application itself. However, facilities for analysing telemetry at scale and in real-time are not available to application users and developers.

As a consequence, we see a proliferation of application- and platform-specific approaches to telemetry management and usage. Due to the inherent complexity of handling telemetry, especially at scale, a lot of effort has to be invested by application developers to integrate telemetry into their applications. This can take focus away from the core mission and purpose of the application and creates a significant redundancy as the wheel is being reinvented over and over again. Another consequence of the complexity is, that only large projects with enough resources and technical expertise can afford to implement telemetry-based optimization and resilience patterns. For a lot of smaller projects, this remains infeasible, which effectively creates a two-class system in which some applications become highly efficient HPC system tenants with a near-optimal return on investment (billed CPU hours), while others, the long tail, are often forced to remain in a suboptimal space. Similarly, systems research and the development and evaluation of novel resilience and optimization patterns and techniques are confined to a small group of researchers with unrestricted access to telemetry which often goes hand-in-hand with elevated privileges on HPC systems.

Supporting the long tail of applications by making telemetry more accessible is one of the main opportunities that lie within this work. If more applications can benefit from telemetry-driven optimization and resilience patterns, both, the efficiency and ultimately the scientific productivity of the individual applications can be increased. This would lead to an overall efficiency increase for HPC systems by reducing unproductive utilization of resources that are caused by hardware and software failures, human error, suboptimal resource usage, and inadequate adaptation to new architectures.

We address the challenges above to lower the barrier for wider adoption of existing HPC telemetry usage patterns. It will stimulate research into novel resilience and optimization patterns which will ultimately lead to more productive utilization of HPC systems. We propose the following approach:

(i) Develop an open, extensible telemetry data model that provides a structural and semantic context for the organization of application and platform telemetry.

(ii) Apply the data platform approach to HPC telemetry and design and build a software platform that addresses the accessibility, availability, and integration challenges.

(iii) Provide an integrated and scalable data analytics and machine learning environment that allows for descriptive, predictive, and prescriptive analysis of telemetry and encourages decoupling it from core HPC application logic.

(iv) Enable a collaborative ecosystem in which telemetry data and derived patterns and services can be shared.

We call this solution a *telemetry platform*. Figure 1.1 provides a high-level overview of the approach which draws heavily on the idea of data platforms that can be found across many industries and large research projects where a central data repository, or data lake, allows business or research teams to develop and share new insights and actions using everything from simple time seriesanalysis to complex statistical and machine learning methods. Data platforms centralize and hide many of the inherent complexities of data management and usage from users, for example, data ingestion, clean-up, storage, cataloguing, access management, and providing appropriate and scalable analytics resources. Given the volume, velocity, and variety of telemetry on a large-scale HPC system and the diverse user groups and use cases, the data platform model appears as an appropriate approach for handling telemetry on HPC systems. Consequently, this thesis aims to answer the following research question:

> *How can the data platform paradigm be effectively applied to telemetry management and usage on HPC systems so that it aids a more efficient development and research workflow, and lowers the barrier for adoption of optimization and resilience techniques in applications and enables novel systems research while remaining feasible to be supported by HPC platform operators?*

In order to answer this research question, the requirements and characteristics of HPC telemetry must be thoroughly understood from the perspective of HPC system operators, application developers and users, and system researchers. This thesis thus seeks to provide a comprehensive definition of HPC telemetry and an overview of use cases and application areas. It is the first work on HPC telemetry addressing existing challenges through a platform paradigm. Thus, it extends existing research in systems monitoring, telemetry data formats, models, ontologies, integration patterns, and analysis with an overarching platform framework.

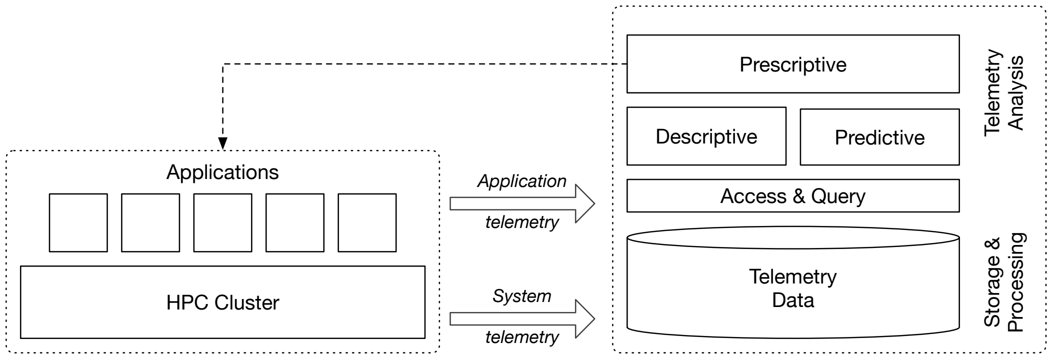


Figure 1.1: An HPC telemetry platform (right) provides storage, access, analytics, and integration capabilities for telemetry data. It provides an integrated environment for telemetry management and usage and enables decoupled application architectures.

1.2 Research Outline

As our point of departure, we look at the diverse viewpoints and requirements of three, largely distinct communities in the HPC ecosystem: platform providers and operators, application developers and users, and HPC researchers. We posit that moving away from insular monitoring and telemetry solutions and towards a more homogeneous system and data repository that is seamlessly integrated with the HPC system, not only benefits the individual communities but also generates further synergetic effects between them. We argue that a telemetry platform should not only provide the raw computing and storage resources but also provide the capabilities and services that help operators, application users, and researchers use these resources most efficiently and cost-effectively. A rich repository of telemetry data is the foundation for any optimization and hence must become part of the core HPC system. This research provides the high-performance computing community with guidance and a better understanding of:

1. The spectrum of telemetry use cases in HPC operations, applications, and research and how all three communities can mutually benefit from a plurality of data in a homogeneous telemetry platform.

2. How to design and implement a data model that can capture telemetry together with its semantic context and the time-variant structure of HPC systems and applications.

3. How to design, implement and operate a scalable data platform as a service and how to integrate it with existing HPC systems.

4. How to approach optimization and resiliency implementations in HPC applications and how to create a more open and sharing research and engineering ecosystem around a telemetry platform.

At the time of this research, a comprehensive telemetry management system that spans both, system and application space does not exist, even though the number of applications and services that require access to telemetry to operate efficiently is increasing. This makes this research important and timely.

1.2.1 Approach and Methodology

To explore the overarching research question "How to design a telemetry platform and how it can be integrated with existing HPC systems and applications", this research adopts a combined qualitative and quantitative research methodology that consists of five steps: (1) use case and requirement analysis; (2) information-model design and validation; (3) conceptual design and architecture of the system; (4) system implementation and integration; and (5) experiments and use case validation. As with every system design, the process is not linear but circular: evaluation results are used to refine system definition and design, implementation and integration scenarios are adapted. While this thesis presents the results linearly, many iterations were done throughout the course of this research in order to arrive at the conclusion presented (figure 1.2). The detailed approach of the five steps is as follows:

1. To understand how to design and build a telemetry platform, we need to understand how telemetry is used on today's HPC systems. Therefore, we conduct a use case study of how telemetry is used across platform operations, application development and operation, adaptive and autonomous system design, federated HPC, and systems research. This gives us both, an in-depth understanding of the state of telemetry management and a set of requirements for designing a data model, telemetry service and interface, and an integration architecture.

2. With the use cases and requirements defined, we can design the telemetry data model. We split the model design process into two parts: first, we define an abstract, formal model that can capture the structure and semantics of any HPC system and application architecture. Then we take the formal definition and define a concrete instantiation of the model, aimed at a typical HPC context, but with a use case and scope-specific architecture, focus, and level of detail in mind.

3. With the use cases and requirements defined, we can also define the conceptual design and architecture of the telemetry platform. Conceptual means that we discuss components, interfaces and their interactions on an implementation-independent level. We decouple the concept of a telemetry platform from its implementation, i.e., the solution space for its functional- from the solution space for its non-functional requirements as the latter potentially change with the implementation context while the former will not.

4. To test the applicability of the telemetry platform concept, we apply the conceptual design and architecture in practice. We define and set up an implementation context that resembles a typical HPC environment. With this testbed in place, we develop a prototype implementation we call SEASTAR.

5. To demonstrate the capabilities of SEASTAR in practice, we implement an existing application that uses machine learning on telemetry data to detect execution anomalies during its execution.

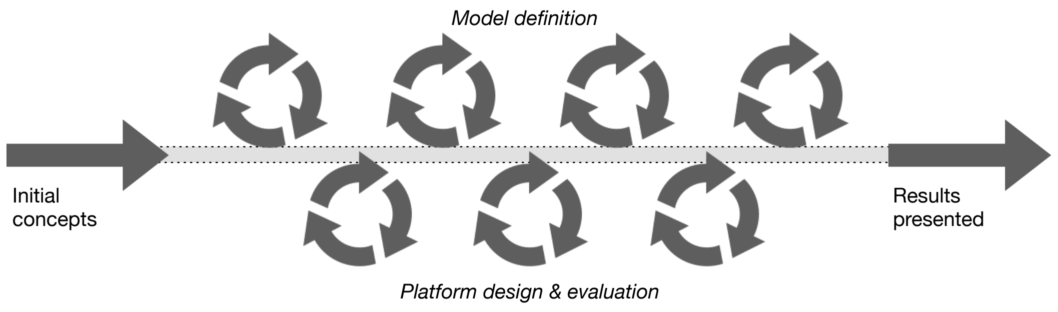


Figure 1.2: As opposed to its linear presentation in this thesis, the model definition and system-design process follows a much more iterative process in practice.

1.2.2 Scope and Limitations

The main goal of this research is to lay out an end-to-end concept for telemetry management and usage on HPC systems, and a clear path for the design, implementation, and integration of it. It does not provide full, production-ready implementation of the proposed telemetry platform as this would be beyond a feasible scope. Instead, this work proposes a conceptual architecture and a specific proof-of-concept prototype implementation of it aiming at a specific implementation context. Both can be used as blueprints and inspiration to build such a system in a production environment. Similarly, we do not provide a comprehensive experimental evaluation and validation of the data platform concepts. However, an evaluation of the central components along with a discussion of the known properties of the architectural decisions builds initial confidence in the feasibility of the platform and approach and provides a predictor for the scalability, cost and ease of integration of the overall approach.

While this research presents a technical solution to our understanding of the telemetry problem, we are very much aware of another, more structural, non-technical problem that our work cannot address. [Allcock et al., 2011] summarizes it quite poignantly:

> Despite everything said above, the real solution to the problem [*the challenges in HPC monitoring*] is sociological. We need champions who are willing to invest time and effort into driving this. An active community needs to be formed. That community needs to make itself heard, convince funders that there is work here worthy of programmatic funding. Present vendors with a set of directions and requirements gained through community consensus. Without that, we will simply continue to talk about what works and what does not and continue to use home-grown, fragmented, sub-optimal, human resource-intensive solutions.

We hope that the results of our research will continue to stimulate and contribute to this ongoing debate and help the community to make a strong point for the future development of telemetry platforms as integral parts of the HPC ecosystem and user and developer experience.

1.3 Novelty and Contributions

This research contributes to the fields of high-performance and distributed computing, particularly to the fields of HPC system and application architecture as well as system and application monitoring. The main contributions are:

1. **Telemetry Platform Paradigm**
   Our main contribution to the field of HPC monitoring and telemetry management is an application of the data platform concept to HPC telemetry and its use cases. While existing telemetry and monitoring solutions focus on collecting telemetry data, our approach focuses on the usability of telemetry and makes analysis an integral part of the overall system instead of locating it externally. This makes the telemetry platform approach a conceptually novel approach to

handling telemetry on HPC systems and distinguishes our solutions, from other, existing approaches. Furthermore, our approach caters to platform operators, application developers and users, and researchers alike. This cross-domain approach has not been taken on by existing research which generally focuses more narrowly on specific user groups or use cases.

2. **Telemetry Information Model**
   A novel telemetry data model that addresses several existing challenges of working with telemetry data by providing a time-variant structural framework, the telemetry graph, in which telemetry data can be organized and localized in a standardized way. To our knowledge, capturing the dynamic structure of and interaction between HPC platform and application components together with the telemetry data they generate has not been proposed and implemented before.

3. **Telemetry Platform Architecture**
   A blueprint and prototype of a concrete implementation and integration architecture of the telemetry platform paradigm and telemetry data model. Based on existing state-of-the-art data platform implementations, we illustrate, how a telemetry platform can be realized using a combination of open-source software components and public cloud building blocks and services. This presents a novel approach, and we have not come across solutions that use this approach for storing and managing HPC telemetry data.

4. **Decoupled Application Architectures**
   A proposal for decoupled HPC application architectures, separating telemetry data management and logic from the core application code. We illustrate how this architecture pattern allows lower-complexity application code and enables the reusability of resilience and optimization capabilities that would otherwise often be tightly coupled to a specific application. We show by example how a machine learning-based, application anomaly detection service can be realized using a decoupled architecture approach. To our knowledge, this research is the first to formalize decoupled application architectures specifically in a telemetry-driven HPC application context.

A few smaller contributions are made at the periphery. These include a use cases survey and requirements gathering for HPC telemetry usage along with an analysis of challenges and opportunities. To our knowledge, this has not been done systematically in existing research. Furthermore, the implementation discussion provides some valuable insights into designing, building and integrating HPC system services in the Amazon Web Service (AWS) public cloud, a topic that has not been widely published about.

1.4 Publications

The publications listed in this section are either a direct outcome of our research or have motivated and contributed to the thought process around HPC telemetry platforms.

1.4.1 Principal Contributions

(i) (2017) O. Weidner, M. Atkinson, and A. Barker. *Seastar: A Comprehensive Framework for Telemetry Data in HPC Environments*. Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2017) held in conjunction with the 26th International Symposium on High Performance Distributed Computing (HPDC 2017), Washington D.C., USA.

This conference paper lays out our concept of a telemetry platform service. It introduces Seastar, a conceptual model and a software framework to collect, store, analyse, and exploit streams of telemetry data generated by HPC systems and their applications. It shows how such a system can be integrated with HPC system architectures and how it enables the swift adoption of common application execution strategies.

(ii) (2016) O. Weidner, M. Atkinson, R. Filgueira Vicente, and A. Barker. *Rethinking High Performance Computing Platforms: Challenges, Opportunities and Recommendations*. Proceedings of the 7th International Workshop on Data Intensive Distributed Computing (DIDC 2016) held in conjunction with the 25th International Symposium on High Performance Distributed Computing (HPDC 2016), Kyoto, Japan.

This conference paper takes a critical look at the dominant HPC ecosystem model and describe the challenges it creates for 2nd generation applications because of its asymmetric resource view, interfaces and software deployment policies. It makes recommendations for an extended, more symmetric and application-centric HPC ecosystem model that adds decentralized deployment, introspection, bidirectional control and information flow and more comprehensive resource scheduling. It describes an early prototype of a non-disruptive implementation of a more symmetric telemetry and platform API based on Linux Containers (LXC) [1].

1.4.2 Supporting Contributions

(i) (2016) V. Balasubramanian, A. Treikalis, O. Weidner, S. Jha. *Ensemble Toolkit: Scalable and Flexible Execution of Ensembles of Tasks*, 45th International Conference on Parallel Processing (ICPP 2016), pp. 458 – 463, Philadelphia, PA, USA. 2016

This conference paper explores the execution of workflow task ensembles on modern HPC systems using overlay scheduling techniques. Working with tens of thousands of concurrent tasks on large-scale infrastructure requires sophisticated telemetry collection approaches, both to analyse and evaluate the experiments as well as to guide the execution of the overlay scheduling framework itself. The research conducted in the paper influenced this thesis as it raised the (painful) awareness of the complete absence of any systematic support for telemetry management on very large HPC systems in the Top500 list.

(ii) (2015) A. Merzky, O. Weidner, and S. Jha. *SAGA: A Standardized Access Layer to Heterogeneous Distributed Computing Infrastructure*. SoftwareX, Volumes 1–2, Pages 3-8, ISSN 2352-7110. 2015

This journal paper disseminates close to ten years of conceptual and practical work on programming interfaces and standards for heterogeneous distributed computing infrastructure. This work has been hugely influential on our research as it has provided us with an in-depth understanding of how standardized in-

[1]After its publication, this paper was picked up by *HPCWire*, a popular news and information resource covering HPC-related topics [HPCWire, 2017] and subsequently highlighted on the Communications of the ACM (CACM) *Tech News* blog [CACM, 2017].

terfaces and abstractions can unlock productivity and cross-fertilization across different scientific domains. In many ways, our work on telemetry platforms is a continuation of this journey.

(iii) (2013) B. Radak, M. Romanus, E. Gallicchio, T. Lee, O. Weidner, N. Deng, P. He, W. Dai, D. York, R. Levy, S. Jha. *A Framework for Flexible and Scalable Replica-Exchange on Production Distributed CI.* XSEDE '13 Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, ISBN: 978-1-4503-2170-9, 2013

Similar to ((i)) this HPC software engineering paper contributed to our research as it also raised the awareness of the absence of systematic support for telemetry management on mainstream HPC systems.

(iv) (2012) A. Luckow, M. Santcroos, O. Weidner, A. Merzky, S. Maddineni, and S. Jha. *Towards a Common Model for Pilot-Jobs.* Proceedings of the 21st ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012), Pages 123-124 New York, NY, USA.

*This conference paper introduces a well-defined, conceptual model for a distributed computing abstraction. It provides a minimal but complete model of Pilot-Jobs (the P*Model), establishes the generality of the model by mapping it to existing Pilot-Job systems, and demonstrates its practical applicability across multiple HPC systems via the Pilot-API. This approach is very similar to and has influenced the model and systems design approach we take in this thesis.*

(v) (2012) A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha. *P*: A Model of Pilot-Abstractions.* IEEE 8th International Conference on E-Science (e-Science 2012), pp. 1-10., Chicago, IL, USA.

*This conference paper extends the research done in ((iv)) by further validating the implementation of the Pilot-API by using multiple distinct Pilot-Job frameworks concurrently across multiple HPC and distributed systems. Furthermore, this paper explores how the P*Model can be applied to distributed data.*

1.5 Thesis Structure

The remaining chapters of this thesis are structured as follows:

- **Chapter 2 — Background and Rationale**
  This chapter looks at the state of telemetry usage and management in HPC. It identifies the main sources of telemetry data across systems and applications and the approaches and systems that are in use today to extract and collect it. Based on these insights it identifies the five major challenges in the field of HPC telemetry: accessibility, availability, integration, structure and semantics, and processing and analysis. It then moves on to a cross-cut through four important telemetry application areas: systems operation, adaptive application architectures, application development, and system research. For each area, it discusses research and practices along with concrete examples of the application and tools used.

- **Chapter 3 — Telemetry Platform**
  This chapter introduces the data platform paradigm, an approach and conceptual architecture that has been shown to enable the efficient management and exploitation of large volumes of continuously generated data across many industries and research applications. It then set the scene for the main hypothesis of this thesis by discussing how the data platform paradigm can be applied to HPC telemetry and the opportunities that emerge from this in the areas of HPC systems research, decoupled and reusable architectures, resiliency and optimization, and application and service architectures that build on machine learning techniques. The second half of chapter 3 provides a requirement analysis for a telemetry platform and presents a conceptual design that covers usage modes, data models, interfaces and integration with HPC systems. The conceptual design serves as the base for the prototype implementation presented in Chapter 5.

- **Chapter 4 — Telemetry Data Model**
  This chapter introduces a telemetry data model that is based on a time-variant labelled multigraph to represent the evolving and changing structure of HPC platform, applications, and the relationships between the two. It discusses the high-level design concepts and provides a formal definition of the telemetry data model and how the requirements defined in chapter 3 are reflected in them. The second half of chapter 4 discusses routes for implementing the data model and

presents a hybrid database design combining and spreading the data across a graph- and a time-series-database.

- **Chapter 5 — Implementation and Evaluation**
  This chapter discusses SEASTAR, a prototype implementation of the telemetry platform concept and its integration with an HPC cluster. It starts by describing the SEASTAR telemetry platform, which is built around the telemetry graph implementation introduced in the previous chapter and realizes telemetry platform capabilities at scale. SEASTAR is built on the AWS public cloud which we use both for SEASTAR and the HPC cluster testbed that we have built to explore the integration between telemetry platforms and HPC clusters. The second part of chapter 5 focuses on the details of this development environment and integration. In the last part of the chapter, we re-visit and implement a use case from chapter 2, an application advisory service that uses machine learning techniques to detect runtime anomalies.

- **Chapter 6 — Conclusion and Future Work**
  In the last chapter, we summarize our work and contributions and reflect on our first experiences with the telemetry platform prototype. We discuss current shortcomings and limitations of our approach and identify areas of future research.

1.6 Summary

We have started this chapter with a short introduction to HPC telemetry, telemetry management, and analytics. We have then looked at the role of telemetry on HPC systems from four viewpoints: system operator, application user, application engineer, and systems researcher and highlighted the strategic importance of telemetry to all four groups of users, particularly in the context of growing HPC system and application size and complexity. We have briefly introduced the challenges observed with telemetry management on existing HPC systems and how they can hamper productivity and performance. We have then moved on to introduce our research statement which aims to address the identified challenges by applying the data platform concept to HPC telemetry and introducing telemetry platforms as an essential component of HPC environments. Lastly, we have described our five-step research approach and methodology, the scope and limitations of this work, and provided an overview of our own publications that have either, led to or were the direct results of this research.

In the next chapter, we will look at the state of telemetry usage and management in HPC. We will discuss the current challenges in more detail and formally introduce the telemetry platform concept along with the benefits that we expect will arise from its adoption.

Chapter 2

Background and Rationale

In this chapter, we discuss the state, application areas and current challenges of telemetry in high-performance computing and motivate the *telemetry platform* paradigm as a way to address them. We begin the chapter with an introduction to the two main categories of telemetry that occur in HPC systems: system telemetry and application telemetry, each illustrated with practical examples. Next, we provide a cross-cut through three important telemetry application areas: systems operation, adaptive application architectures, and application development. For each area, we provide a brief introduction of research and practices along with concrete examples of the applications and tools used. We analyse them to understand how they extract, collect and use telemetry. Based on these insights, we identify five major challenges in the field of HPC telemetry: accessibility, availability, integration, structure and semantics, and processing and analysis.

2.1 HPC Telemetry

We define telemetry as the continuous stream of time seriesand event data that is generated on HPC systems by the hardware, operating systems, services, runtime systems, and applications. Telemetry is generated continuously during the operation of an HPC system. Temperature and power consumption readings from hardware sensors, job and process metrics, network and disk I/O metrics, and Message Passing Interface (MPI) message size metrics are all examples of telemetry data generated on an HPC system. Telemetry provides insights into the current and past states of an HPC system and the applications running on it. It also provides the foundation to make predictions about their future behaviour. We distinguish between two distinct sources of telemetry: sys-

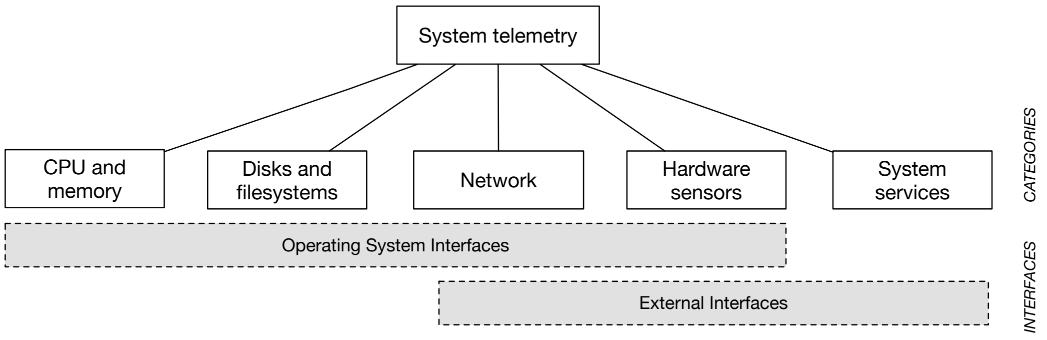


Figure 2.1: A high-level taxonomy for system telemetry that distinguishes between five different functional areas and between operating-system and external interfaces as telemetry sources.

tem telemetry, and application telemetry. System telemetry is continuously generated during the normal operation of an HPC system by a variety of components: the operating systems commanding the compute-, storage-, networking- and utility-nodes, environmental sensors that monitor power consumption, temperature, and other external factors, and HPC system services such as job queueing and object storage systems. Combined, we call these sources system telemetry. In contrast to that lies application telemetry. We define application telemetry as the information that is generated during the execution of applications. This includes information about an application's system resource allocation and interaction as well as telemetry information generated by the application itself, such as internal performance metrics of an adaptive algorithm. Next, we provide an overview of system and application telemetry and their sources.

2.1.1 System Telemetry

We use the simple taxonomy shown in figure 2.1 to structure the system telemetry space into five functional areas: CPU and memory telemetry, disk and filesystem telemetry, network telemetry, hardware sensor telemetry, and system services telemetry. Telemetry can be sourced either directly from operating-system interfaces, or via other, external interfaces. The dominant source of system telemetry is the operating-system controlling the individual nodes of an HPC system. Depending on the size of an HPC cluster, the number of compute nodes can range from tens to tens of thousands of nodes.

Operating systems continuously collect telemetry about the state of the hardware they run on and the processes they are running. This information is critical for operating-system functions such as scheduling, error handling and performance optimization. Telemetry provided by the operating-system is generally ephemeral. Beyond basic aggregated statistics, it is usually not possible to query the state of a process or operating-system metric at a previous instant in time. If the data is not captured and made persistent, it is lost. Operating systems can provide multiple interfaces to telemetry, typically a kernel-level programming interface and one or more higher-level user-space interfaces and tools. On Linux, one common user-space interface to operating-system telemetry is the Process Filesystem (ProcFS) [Faulkner and Gomes, 1991]. ProcFS is a hierarchically structured virtual file system (typically mounted at `/proc`) that exports data about the state of the operating-system, including system- and process-level telemetry about memory, CPUs, disks, and file systems. In this section we use ProcFS as a guide for exploring the different telemetry metrics available on the operating-system-level. Other UNIX and non-UNIX operating systems provide similar or equivalent facilities. [Juve et al., 2015] provides a good overview of operating-system-level telemetry sources and the tools to extract them.

In our taxonomy we summarize other sources of telemetry that are not under direct control or exposed by the operating-system as external interfaces. External interfaces can be anything from flat files containing data points to API endpoints. The telemetry of external hardware components such as network equipment is usually accessible via external interfaces, so is the telemetry of hardware sensors like external climate and power consumption sensors. System services, such as batch schedulers or database servers are another category of system telemetry sources that expose data via external interfaces.

CPU and Memory Telemetry

CPU telemetry contains node-level metrics about the individual processor cores, number of context switches, and the time the cores have spent in user, nice, system (kernel), idle, iowait, irq, and softirq modes. Listing 1 shows an example of how these metrics are exposed through the `/proc/stat` interface. The meaning of the columns are as follows, from left to right, (1) normal processes executing in user mode, (2) niced processes executing in user mode, (3) processes executing in kernel mode, (4) idle, (5) waiting for I/O to complete, (6) servicing interrupts, and (7) servicing softirqs. These metrics can be used to understand the overall load on compute nodes and how much time the individual CPU cores spend in user and in kernel mode, which can be an important indicator for how efficiently application processes are executed. High-level information about the type of CPUs and their properties are available through the `/proc/cpuinfo` interface.

```
cpu  2255 34 2290 22625563 6290 127 456
cpu0 1132 34 1441 11311718 3675 127 438
cpu1 1123 0 849 11313845 2614 0 18
intr 114930548 113199788 3 0 5 263 0 4 [...]
ctxt 1990473
btime 1062191376
[...]
```

Listing 1: Example output (truncated) of `/proc/stat` showing a snapshot of CPU time spent in user, nice, system, idle, iowait, irq, and softirq modes.

Similarly, memory telemetry provides information on a node's memory usage. The `/proc/meminfo` interface (listing 2) provides insights into how much memory is currently available, free, buffered, and cached. Additional information about page sizes, and counts, free and purgeable pages, swap I/O are available through other interfaces. Memory metrics are an important component of system telemetry as they provide valuable insights into the memory pressure a node is experiencing during operation. Table 2.3 at the end of this section provides an overview of common CPU and memory metrics available via operating-system interfaces.

```
MemTotal:        5564912 kB
MemFree:         4109724 kB
MemAvailable:    4759432 kB
Buffers:          205200 kB
Cached:           642992 kB
Hugepagesize:       2048 kB
[...]
```

Listing 2: Example output (truncated) of `/proc/meminfo` showing a snapshot of total, free, and available node memory.

Disk and Filesystem Telemetry

Disk I/O and filesystem telemetry provide insights into system-wide I/O statistics, the utilization and performance of a node's local and remote filesystems and physical hard disks. Table 2.1 at the end of this section provides an overview of common Disk I/O and filesystem metrics available via operating-system interfaces. Listing 3 shows an example of per-disk I/O statistics available through the `/proc/diskstats` interface. The meaning of the columns are as follows, from left to right, (1) major number, (2) minor number, (3) device name, (4) reads completed successfully, (5) reads merged, (6) sectors read, (7) time spent reading (ms), (8) writes completed, (9) writes merged, (10) sectors written (11) time spent writing (ms), (12) I/Os currently in progress, (13) time spent doing I/Os (ms), and (14) weighted time spent doing I/Os (ms)

```
8       0 sda 174 0 10666 284 0 0 0 0 0 236 284
8       1 sda1 103 0 8306 188 0 0 0 0 0 172 188
8      16 sdb 30896 64 1152590 15168 12027747 2049451 352088760 25764004 0 [...]
8      17 sdb1 298 0 10756 76 199 0 749016 80 0 120 156
8      18 sdb2 4 0 8 0 0 0 0 0 0 0 0
8      21 sdb5 30528 64 1138578 15084 11870547 2049451 351339744 25692816 0 [...]
252       0 dm-0 30377 0 1130498 15040 13933233 0 351339744 27855208 0 [...]
252       1 dm-1 137 0 6528 24 0 0 0 0 0 12 24
```

Listing 3: Example output (truncated) of `/proc/diskstats` showing a snapshot of per-disk I/O operations currently in progress, completed reads and writes, and time spent reading and writing.

Network Telemetry

Network telemetry plays a particularly important role in high-performance computing as the majority of parallel, tightly-coupled applications, such as applications based on MPI [Gropp et al., 1996] and OpenMP [Dagum and Menon, 1998], rely heavily on efficient network communication. Furthermore, since most HPC systems use shared network filesystems to make data available across nodes, network telemetry provides valuable insights into and help to understand file I/O performance on these shared filesystems. For example, telemetry of a completely saturated network interface can help explain why the read/write performance of a specific application process stays below the average. Network telemetry can be extracted from two main sources: on the node-level from the operating-system and on the network-level from network hardware components like switches and interconnect technologies. Table 2.2 gives an overview of some common network metrics, available on node-level. Listing 4 gives an example of common node-level network metrics exposed via the `/proc/net/dev` interface, such as bytes and packets sent and received, and various transmission error counters.

```
Inter-|   Receive
face |bytes    packets errs drop fifo frame compressed multicast
   lo: 584717770 1024626    0    0    0     0          0
enp3s0: 76479726  193811    0    0    0     0          0

|  Transmit
|bytes    packets errs drop fifo colls carrier compressed
0 584717770 1024626    0    0    0     0       0   0
0 99226362  137064    0    0    0     0       0   0
```

Listing 4: Example output of `/proc/net/dev` showing a snapshot of per-interface network I/O statistics such as packet read/write operations, errors, and dropped packages.

Network telemetry is used by HPC system operators to identify and mitigate network congestions that can occur due to faulty hardware and software, but also due to unexpected, problematic application resource usage patterns. HPC applications use network telemetry in many ways. In [Jha et al., 2007b] for example, we use wide-area and intra-cluster network telemetry to decide where to place workloads that require large bandwidth to stage-in large data volumes prior to execution. In another example, [Filgueira et al., 2010] use intra-node network telemetry to decide when to apply an MPI compression algorithm.

Hardware Sensor Telemetry

Hardware sensors provide metrics about the physical operation environment of an HPC system, such as temperature and power consumption. Most hardware components of typical HPC nodes have multiple environmental sensors built in: modern hard disks, mainboards, CPUs, and GPUs are typically equipped with one or more temperature and voltage sensors. These metrics are used in node-internal feedback loops to control cooling facilities such as CPU and chassis fans, and, if a critical temperature threshold has been exceeded, shut down or throttle hardware components to prevent them from terminal failure. Most telemetry from internal hardware sensors is available through operating-system (kernel) interfaces and made available through user-space tools and libraries, such as the `lm-sensors` packages [Lysoněk, 2019]. Listing 5 shows a sample output.

Beyond node-level sensors, many HPC systems and data centres deploy external thermal and power consumption sensors. These sensors are often used for energy-

```
coretemp-isa-0000
Adapter: ISA adapter
Core 0:      +41.0°C  (high = +78.0°C, crit = +100.0°C)

coretemp-isa-0001
Adapter: ISA adapter
Core 1:      +41.0°C  (high = +78.0°C, crit = +100.0°C)

w83627dhg-isa-0290
Adapter: ISA adapter
Vcore:       +1.10 V  (min =  +0.00 V, max =  +1.74 V)
in1:         +1.60 V  (min =  +1.68 V, max =  +1.44 V)   ALARM
AVCC:        +3.30 V  (min =  +2.98 V, max =  +3.63 V)
VCC:         +3.28 V  (min =  +2.98 V, max =  +3.63 V)
temp1:       +36.0°C  (high = +63.0°C, hyst = +55.0°C)
temp2:       +39.5°C  (high = +80.0°C, hyst = +75.0°C)
temp3:      +119.0°C  (high = +80.0°C, hyst = +75.0°C)
```

Listing 5: Example output (truncated) of the `lm-sensors` command-line tool listing current temperature and voltage levels for various system components.

optimization techniques, such as turning on/off machines, power-aware consolidation algorithms, and machine learning techniques to deal with uncertainty while maximizing performance [Barroso and Hölzle, 2007, Bianchini and Rajamony, 2004, Berral et al., 2010].

System Services

The last category of system telemetry sources are HPC system services. These can include workload managers, databases, network attached storage, object stores, and hypervisor and container orchestrators. Each of these services expose a unique set of metrics that can help users to understand and interpret the systems state and behaviour.

System service telemetry is generally exposed to service-specific APIs, libraries and command-line tools. The SLURM workload manager [Yoo et al., 2003] for example, provides a number of different command line tools that provide insights into the state of HPC cluster nodes and jobs. The `sinfo` command for example returns the status of all HPC cluster nodes and partitions (listing 6). Similarly, the `sacct` and the `squeue` commands provide detailed information about the status of job and queues. HPC system operators often collect this information in order to compile usage statistics which in turn serve as the foundation for usage optimization. If for example, queues

```
PARTITION  AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*        up    6:00:00     24   idle a1a-u2-c10-b8,a1a-u2-c11-b[2-8],...
small         up 1-00:00:00     24   idle a1a-u2-c10-b8,a1a-u2-c11-b[2-8],...
medium        up 7-00:00:00     24   idle a1a-u2-c10-b8,a1a-u2-c11-b[2-8],...
large         up 365-00:00:     22   idle a1a-u2-c10-b8,a1a-u2-c11-b[2-8],...
gpu           up 365-00:00:      1   down* gpu01
gpu           up 365-00:00:      4   idle gpu[02-05]
infiniband    up 365-00:00:      2   idle bc02bl[03,12]
```

Listing 6: Example output (truncated) of the SLURM `sinfo` command-line tool showing the current status of all HPC cluster nodes and partitions.

Metric	**Description**
`disk_total`	total space available on device.
`disk_used`	used space on device.
`disk_free`	free space available on device.
`disk_io_read_count`	number of reads.
`disk_io_write_count`	number of writes
`disk_io_read_bytes`	number of bytes read.
`disk_io_write_bytes`	number of bytes written.
`read_io_time`	time spent reading from disk.
`write_io_time`	time spent writing to disk.
`busy_io_time`	time spent doing actual I/Os.

Table 2.1: Overview of common operating-system level I/O and filesystem telemetry.

for certain job sizes and durations are consistently underutilized, an operator might choose to adjust its size or configuration accordingly. Continuously collecting scheduler information is vital for that. However, not only system operators use scheduler telemetry, but also adaptive applications extract and use this information. In [Balasubramanian et al., 2016] for example, we use queue status and statistics to adaptively schedule workloads across multiple HPC clusters.

Metric	Description
net_io_bytes_sent	number of bytes sent
net_io_bytes_recv	number of bytes received
net_io_packets_sent	number of packets sent
net_io_packets_recv	number of packets received
net_io_errin	total number of errors while receiving
net_io_errout	total number of errors while sending
net_io_dropin	total number of incoming packets which were dropped
net_io_dropout	total number of outgoing packets which were dropped

Table 2.2: Overview of common operating-system level network telemetry.

Metric	Description
load_1	the average system load over the last 1 minutes.
load_5	the average system load over the last 1 minutes.
load_15	the average system load over the last 1 minutes.
cpu_count	number of CPUs.
cpu_core_count	number of cores per CPU.
cpu_freq_max	maximum CPU frequency.
cpu_freq_min	minimum CPU frequency.
cpu_times_user	time spent by normal processes executing in user mode.
cpu_times_system	time spent by processes executing in kernel mode.
cpu_times_system	time spent doing nothing.
cpu_times_nice	time spent by prioritized processes executing in user mode.
cpu_times_iowait	time spent waiting for I/O to complete.
cpu_times_irq	time spent for servicing hardware interrupts.
cpu_times_softirq	time spent for servicing software interrupts.
cpu_times_steal	time spent by virtualized operating systems.
cpu_ctx_switches	number of context switches (voluntary + involuntary) since boot.
cpu_interrupts	number of interrupts since boot.
cpu_softinterrupts	number of software interrupts since boot.
cpu_syscalls	number of system calls since boot.
mem_total	total physical memory.
mem_available	the memory that can be given instantly to processes.
mem_used	memory used.
mem_free	memory not being used and that is readily available.
mem_active	memory currently in use or very recently used.
mem_inactive	memory that is marked as not used.
mem_buffers	cache e.g., for file system metadata.
mem_cached	memory allocated to other caches.
mem_shared	memory that may be simultaneously accessed by multiple processes.

Table 2.3: Overview of common operating-system level CPU and memory telemetry.

2.1.2 Application Telemetry

In contrast to system telemetry, we define application telemetry as the information that is generated during the execution of applications on an HPC cluster. This includes information about an application's system resource allocation and interaction as well as telemetry information generated by the application itself, such as internal performance metrics of an adaptive algorithm. We use the high-level taxonomy from Figure 2.2 to structure this space: process-level telemetry is sourced from the operating-system, while job-level telemetry and telemetry generated by runtime systems and programming frameworks is accessible via external interfaces. Lastly, application-specific telemetry is generated by the application directly.

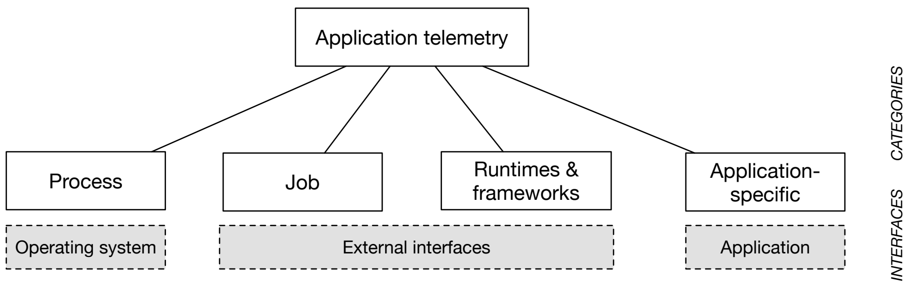


Figure 2.2: A high-level taxonomy for application telemetry. We distinguish between four different functional areas and between operating-system and external interfaces as telemetry sources.

Process-Level Telemetry

Just like system telemetry, process-level telemetry can be extracted via the Procfs virtual filesystem. The information available in procfs varies widely among UNIX systems, but on many systems, including Linux, it provides a directory for each process containing files for different types of information about the process. `/proc/[pid]/stat` for example contains CPU usage information (utime, stime) and current memory usage and `/proc/[pid]/io` (listing 7), contains information about the number of bytes read and written by the process. Table 2.4 at the end of this section gives an overview of some common process metrics.

Another type of process telemetry are hardware performance counters. They can provide more detailed insights into the resources used by a process. Performance counters track the number of hardware operations performed by a CPU core in special-

```
rchar: 5975377879
wchar: 72
syscr: 41988384
syscw: 8
read_bytes: 3948544
write_bytes: 48341413888
cancelled_write_bytes: 0
```

Listing 7: Example output (truncated) of `/proc/[pid]/io` showing a snapshot of bytes read and written by the process.

purpose registers. The types of available counters differs widely between different systems, hardware platforms, and CPUs, but typically there are counters for cycles, instructions, floating-point operations, cache hits, cache misses, branches, loads, stores, and many other CPU operations. PAPI [Browne et al., 2000] is a cross-platform popular library for querying performance counters, and the Linux `perf` tools (listing 8) records performance counters at the process level.

```
        5,099   cache-misses        #      0.005 M/sec (scaled from 66.58%)
        235,384 cache-references    #      0.246 M/sec (scaled from 66.56%)
    9,281,660   branch-misses       #      3.858 %     (scaled from 33.50%)
    240,609,766 branches            #    251.559 M/sec (scaled from 33.66%)
1,403,561,257   instructions        #      0.679 IPC   (scaled from 50.23%)
2,066,201,729   cycles              #   2160.227 M/sec (scaled from 66.67%)
            217 page-faults         #      0.000 M/sec
            3   CPU-migrations      #      0.000 M/sec
            83  context-switches    #      0.000 M/sec
    956.474238  task-clock-msecs    #      0.999 CPUs
[...]
```

Listing 8: Example output (truncated) of the Linux `perf` tool showing hardware counter statistics such as cache-misses and context switches for a process

Job-Level Telemetry

Job-level telemetry consists of the data collected by HPC workload managers during the executing of a job. Typical job-level metrics include job start and stop time, execution time, allocated nodes, cores and memory. Listing 9 shows example job-level telemetry provided by a SLURM workload manager that was integrated with the

NVIDIA Data Centre GPU Manager (DCGM) [1] which provides additional information about the job's GPU usage.

```
|-----  Execution Stats  ------------+-----------------------------------------|
| Start Time                         | Tue Apr  9 20:55:39 2019                |
| End Time                           | Tue Apr  9 21:44:38 2019                |
| Total Execution Time (sec)         | 2938.68                                 |
| No. of Processes                   | 8                                       |
+-----  Performance Stats  ----------+-----------------------------------------+
| Energy Consumed (Joules)           | 4088486                                 |
| Power Usage (Watts)                | Avg: 1380.84, Max: N/A, Min: N/A        |
| Max GPU Memory Used (bytes)        | 16467886080                             |
| Clocks and PCIe Performance        | Available per GPU in verbose mode       |
+-----  Event Stats  ----------------+-----------------------------------------+
| Single Bit ECC Errors              | 0                                       |
| Double Bit ECC Errors              | 0                                       |
| PCIe Replay Warnings               | Not Specified                           |
| Critical XID Errors                | 0                                       |
+-----  Slowdown Stats  -------------+-----------------------------------------+
```

Listing 9: Example output (truncated) of job-level GPU performance information provided by the NVIDIA Data Centre GPU manager and SLURM job manager.

Runtime System and Programming Framework Telemetry

The third category of application telemetry is runtime system and programming framework telemetry. Every programming language and API invoked stand-alone programming frameworks such as POSIX threads [Nichols et al., 1996] or MPI have some form of a runtime system. Runtime system behaviour can be defined as behaviour not directly attributable to the program itself. For example, the runtime system of the C programming language, among others, manages the processor stack and create space for local variables, while the runtime system of an MPI library, among others, manages communication and coordination between distributed processes.

Many runtime systems provide access to telemetry via library functions. The JAVA Virtual Machine (JVM) for example provides continuous statics on internal threads, garbage collection, and memory usage. Another example of software framework telemetry is the resource utilization statistics data generated by MPI. These can

[1] `https://developer.nvidia.com/dcgm`

```
Total job time 2.203333e+02 sec
Total MPI processes 128
Wtime resolution is 8.000000e-07 sec

activity on process rank 0
comm_rank calls 1      time 8.800002e-06
get_count calls 0      time 0.000000e+00
ibsend calls    0      time 0.000000e+00
probe calls     0      time 0.000000e+00
irecv calls     22039  time 9.76185e-01   datacnt 23474032 avg datacnt 1065
send calls      0      time 0.000000e+00
ssend calls     0      time 0.000000e+00
isend calls     22039  time 2.950286e+00
wait calls      0      time 0.00000e+00   avg datacnt 0
waitall calls   11045  time 7.73805e+01   # of Reqs 44078  avg data  cnt 137944
barrier calls   680    time 5.133110e+00
alltoall calls  0      time 0.0e+00       avg datacnt 0
alltoallv calls 0      time 0.000000e+00
reduce calls    0      time 0.000000e+00
allreduce calls 4658   time 2.072872e+01
bcast calls     680    time 6.915840e-02
...
```

Listing 10: Example output (truncated) for a single rank of an MPI program that was run on 128 processors, using a user-created profiling library that performs call counts and timings of common MPI calls

for example be used to determine potential performance problems caused by lack of MPI message buffers and other MPI internal resources. MPI provides access to this statistics via standard library functions that are accessible either directly by applications (listing 10) or via other MPI profiling frameworks and tools such as *mpiP* [Vetter and Chambreau, 2014].

Application-Specific Telemetry

The last category of application is application-specific telemetry. As opposed to the other three application telemetry types which capture application telemetry from an outside perspective and provide the same type of metrics for any application running on an HPC system, application-specific telemetry consists of metrics defined by and relevant to a specific application. An Adaptive Mesh Refinement (AMR) application for example can continuously generate and emit information about grid cell density and distribution across processes in order to understand the progression of a calculation. This information becomes particularly valuable if it is put into the same context with other system and application telemetry as it allows for a more holistic interpretation of application behaviour and performance. A telemetry platform that allows developers to collect and analyse application-specific telemetry within a global framework of reference would significantly simply the process of generating this holistic insight.

Metric	**Description**
`io_read_count`	Number of read operations performed.
`io_write_count`	Number of write operations performed.
`io_read_bytes`	The number of bytes read.
`io_write_bytes`	The number of bytes written.
`cpu_num`	The CPU the process is currently running on.
`cpu_affinity`	The current CPU affinity of the process.
`cpu_user_time`	Time spent in user mode.
`cpu_system_time`	Time spent in kernel mode.
`cpu_iowait_time`	Time spent in kernel mode.
`mem_rss`	The non-swapped physical memory a process has used.
`mem_vms`	The total amount of virtual memory used by the process.
`mem_shared`	Memory that could be potentially shared with other processes.
`mem_dirty`	The number of dirty memory pages.
`mem_swap`	Amount of memory that has been swapped out to disk.
`files_fd`	The file descriptor of an open file.
`files_path`	The absolute name of an open file.
`con_fd`	The socket file descriptor of a connection.
`con_family`	The address family of a connection.
`con_type`	The type (STREAM, DGRAM ...) of a connection.
`con_local_address`	The local address of a connection.
`con_remote_address`	The remote address of a connection.
`con_status`	Status of a connection.
`children`	Subprocesses associated with a process.

Table 2.4: Overview of common operating-system level process telemetry.

2.2 Application Areas

In order to better understand how telemetry is used on HPC systems, we look at three different application areas: *HPC system operations*, *adaptive application architectures*, and *application development*. For each of the three application areas, we identify a number of common research and practices and underpin them with concrete examples of tools and applications. Systems operations, for example has a more monitoring-centric approach to HPC telemetry and concrete examples predominantly focus on a macro-level understanding of performance and stability indicators across all system components and applications. Examples in the application development category on the other hand are more focused on fine-grained application-specific system telemetry, often overlaid with custom application telemetry in order to carry out more micro-level optimizations. In order to gain some better understanding on how telemetry is used, we try to answer the following questions for each of the use cases:

(i) **How is telemetry collected?**
Which interfaces are used, and what is the software architecture that implements the collection mechanisms?

(ii) **Which telemetry is collected?**
Which system and application telemetry are extracted?

(iii) **How is telemetry structured?**
How is telemetry data is formatted, and structured internally?

(iv) **How is telemetry stored?**
Is the collected data ephemeral or is it stored for later use? If so, how is data storage implemented?

(v) **How is telemetry shared?**
Is the collected data accessible only from within the system or is it also shared and accessible via external interfaces?

While the examples in this section do not aim for completeness, they provide enough insights into telemetry usage across the different categories in order to understand common challenges. These challenges match, in different varieties and severities, the challenges we laid out in the previous chapter and hence serve as the input for the requirement analysis that we conduct in the second part of this chapter.

2.2.1 System Operations

The majority of HPC systems are run by dedicated teams of operations staff that are responsible for the continuous operation and optimization of an HPC platform's hardware and software. Systems operation is often driven by a number of key objectives, such as:

- System availability and stability
- System utilization
- User satisfaction

In order to fulfil these objectives, many HPC operations centres organize around three core practice domains: daily operations, strategic planning and optimization, and application and user support as shown in figure 2.3. Daily operations focus on optimizing the availability and stability of the system and detecting and mitigating hardware and software errors. Strategic planning and optimization looks at the long-term utilization patterns of one or more systems and advises on future extensions or reconfiguration of an HPC estate. Lastly, application and user support helps HPC users and developers to develop, deploy, run, and optimize HPC application code and advises on system particularities and best practice. In the following we will look at these three practice domains in more detail and discuss several concrete examples in regard to their use of telemetry.

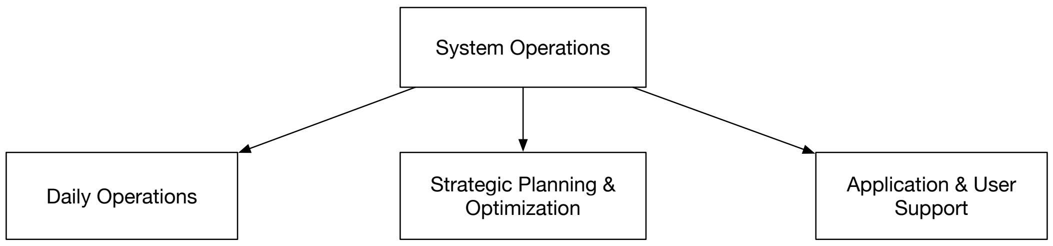


Figure 2.3: HPC system operations organizes around three core practices: daily operations, strategic planning and optimization, and application and user support.

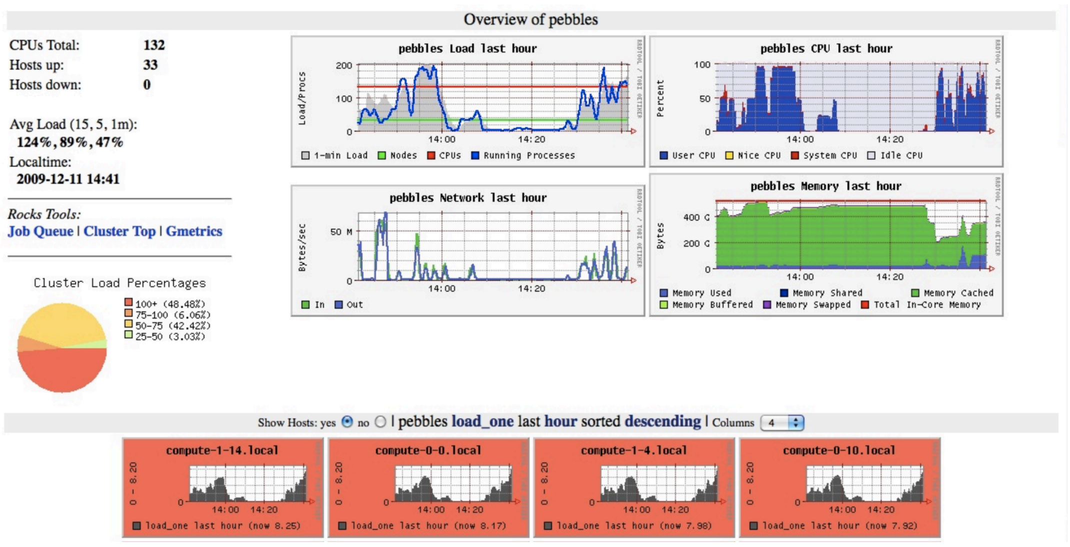
Daily Operations

Monitoring systems collect, and visualize the relevant telemetry. Monitoring is a key capability for HPC operators to ensure system stability and availability. Given the ever-growing size of systems and the associated computational, I/O, and network demands placed on them by applications, failures in large-scale systems become commonplace. To address node failure and to maintain the health of the system, platform monitoring must be able to quickly identify failures so that they can be repaired either automatically or via out-of-band means. In large-scale systems, the interplay amongst computational nodes, network switches and links, and storage devices can be complex. A monitoring system that captures the telemetry characterizing these interactions can often lead to a better understanding of a system's microscopic and macroscopic behaviour. As platforms continuously grow in size and complexity, bottlenecks are likely to arise in various locations. A monitoring system can assist platform operations by providing a global view of the system, which can be helpful in identifying performance problems and, ultimately, assisting in capacity planning.

Monitoring tools are a critical component to ensure stable operation of an HPC systems and have evolved along with HPC systems for multiple decades. Development has been driven mainly by the HPC system operator communities with their particular use cases in mind which evolve around system stability, error detection and optimizing system utilization. Most HPC monitoring tools align around similar, three component architectures that consist of a data collection component, a database to store time seriesdata, and a graphical front-end to visualize the data in different views. In the following, we will look into the details of Ganglia, a popular HPC monitoring system in. Similar systems include *Supermon* [Sottile and Minnich, 2002] and NAGIOS [Josephsen, 2007].

Ganglia [Massie et al., 2004] is based on a hierarchical design and data model which can accommodate not only single HPC systems but also federations of clusters. In order to scale, Ganglia relies only on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. The primary design objective for Ganglia's data structures and algorithms was to accommodate low per-node overheads and high concurrency. It uses the Extensible Markup Language (XML) format for data representation the External Data Representation (XDR) format for portable data transport, and RRDtool [Oetiker, 2017] for data storage and visualization.

Ganglia consists of two services: The Ganglia monitoring daemon (gmond) and the Ganglia Meta Daemon (gmetad). Gmond runs on every node of a cluster and provides monitoring by implementing the listen/announce protocol and responding to client requests by returning an XML representation of its monitoring data. Gmetad provides federation of multiple clusters. A tree of TCP connections between multiple gmetad daemons allows monitoring information for multiple clusters to be aggregated. In addition, Ganglia provides a command-line program (gmetric) that applications can use to publish application-specific metrics and a client-side library provides programmatic access to a subset of Ganglia's features. Gmond publishes two types of metrics, built-in metrics which capture node state and user-defined metrics which capture arbitrary application-specific state, on a well-known multicast address. For built-in metrics, gmond currently collects and publishes about 30 different metrics depending on the operating system and CPU architecture it is running on through well-defined interfaces, such as `/proc`, KVM [2], and kstat. Some base metrics include the number of CPUs, CPU clock speed, CPU (user, nice, system, idle), load (1, 5, and 15 min averages), memory (free, shared, buffered, cached, total), processes (running, total), swap (free, total), system boot time, system clock, operating-system (name, version, architecture), and (maximum transmission unit). User-defined application telemetry can be arbitrary and is not predefined by the system. Telemetry is structured by adding cluster- and node-name metadata that allows a basic understanding of the underlying system architecture.


Overview of pebbles

CPUs Total:	132
Hosts up:	33
Hosts down:	0

Avg Load (15, 5, 1m):
124%, 89%, 47%
Localtime:
2009-12-11 14:41

Rocks Tools:
Job Queue | Cluster Top | Gmetrics

Show Hosts: yes [✓ ●] no ○ | pebbles load_one last hour sorted descending | Columns 4

Figure 2.4: Screenshot of a web-based HPC monitoring dashboard generated by Ganglia.

[2]KVM (Kernel-based Virtual Machine) is a virtualization solution for Linux on x86 hardware containing virtualization extensions.

Telemetry data is stored in RRDtool a fixed-size, circular buffer-based database, designed for storing and summarizing time seriesdata. RRDtool assumes time-variable data in intervals of a certain length. This interval is specified upon creation of an RRD database and cannot be changed afterwards, i.e., it is fixed. RRDtool will automatically interpolate any submitted data to fit its internal time-steps. The interval defines the maximum granularity of data points. Multiple data points can be consolidated according to a consolidation function (e.g., average, minimum, maximum) to form a consolidated data point. Recording telemetry at a granularity beyond what is defined as the time-step interval is not possible. Since the database is constant in size and circular, it will at some point ”wrap around”, meaning that the next insertion will overwrite the oldest entry. Telemetry retention is therefore defined by the time-step interval and size limit the database was created with. The main interface to Ganglia is a web-based visualization frontend. The graphs are pre-generated by the RRDtool backend and displayed through the frontend (figure 2.4). The gmetad service can be configured to listen on a specific TCP port and reply with XML metric data. It can return both metric summaries and the results for simple queries such as specifying a cluster or hostname. However, this feature is disabled by default, which prevents programmatic access to telemetry altogether.

Cray System Snapshot Analyser (SSA) [Duckworth et al., 2017] is a customer service and remote diagnostics application developed by Cray Inc. and designed to support HPC cluster issue diagnosis and reduce time to resolution. SSA’s main goal is to support Cray’s service engineers to remote-diagnose system issues via “snapshots”, system diagnostic information, hardware and software inventories bundled and sent to Cray’s support servers. SSA is typically used on-demand and at the request of Cray customer service. SSA uses a client service running on a dedicated Cray management host to extract system telemetry via Cray’s specific telemetry interfaces. The data is then uploaded to a remote server, where it is stored and further analysed to understand product state and configuration at a given in time, and changes in product state and configuration over time.

Strategic Planning and Optimization

In addition to the everyday concerns of platform operations, HPC system operators also need to consider long-term strategic planning and optimization of their system estate. This includes system hardware upgrades and replacement of obsolete systems but also hard- and software reconfiguration of existing systems to cater for changes in the application and user landscape. Especially the latter is important in order to maintain a high user-satisfaction in a constantly changing and evolving use case and application landscape. One example for system reconfiguration is job queue optimization. workload managers define multiple queues to which jobs can be submitted. These queues are configured to accept jobs of certain types, typically defined by maximum runtime and maximum number of cores allocatable. Queues can furthermore be configured to give access to nodes with specific hardware configurations, such as Graphics Processing Unit (GPU) and Field-Programmable Gate Array (FPGA) subsystems. Queue configurations are initially designed with assumptions made about the expected job workload and type of jobs submitted by the user. Over time,workload and type of jobs can change which can lead to a situation where for example increasing numbers of “bag-of-tasks”-style jobs or jobs with heavy disk I/O are not well-supported by a cluster and queue configuration that assumes massively parallel, compute intensive workloads. Many tools exist to support data centre operators with strategic insights into application behaviour and long-term changes in job mix. Open XDMoD [Palmer et al., 2015] for example, creates a telemetry data warehouse by ingesting the HPC centre’s resource manager, telemetry quality-of-service metrics and job-level performance telemetry. It provides a rich set of analysis and charting tools (figure 2.5) that let system operators quickly display a wide variety of job accounting metrics over any desired timeframe. It enables the comprehensive management of HPC resources, allowing HPC centre personnel to ensure that the resource is operating efficiently and to determine what type of applications are running, how efficiently they are running, and what resources they’re consuming, all of which are important to strategic optimization of the system.

Increasingly advanced technologies are applied to telemetry in order to classify jobs of unknown application, characterize the job mixture, and harness the variation in node and time dependence for further analysis. In [Gallo et al., 2015] for example, machine learning techniques were applied to XDMoD job accounting and performance data for application classification. The results demonstrate that community applications have characteristic signatures which can be exploited for job classification.

Another key objective for strategic planning and optimization is to improve the energy efficiency of HPC systems. Since energy is a major cost item for data-centres, and becoming a dominating factor for the total cost of ownership (TCO) over the lifetime of an HPC system, energy efficiency has become a major research area not just

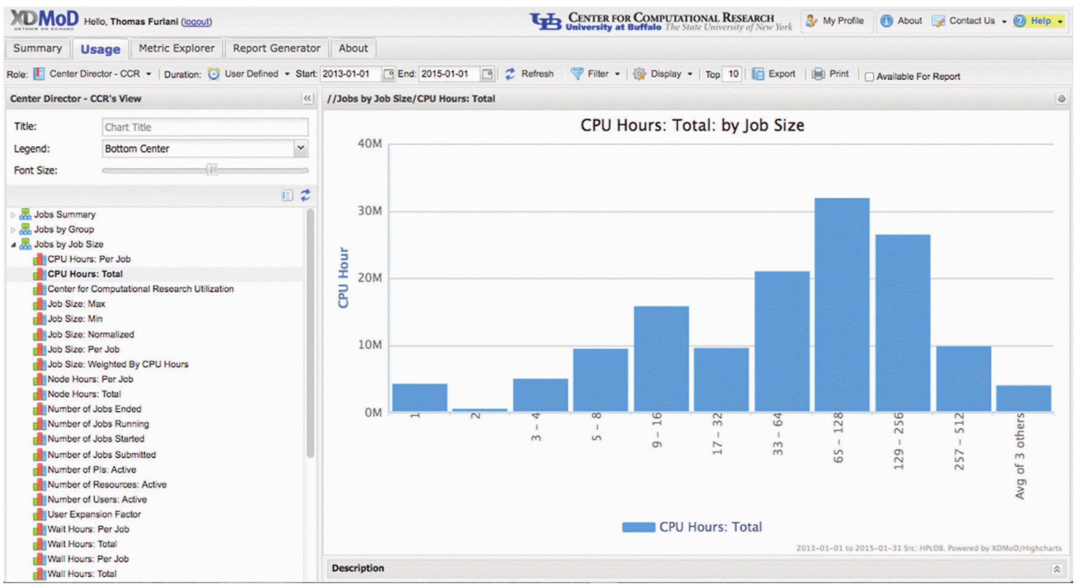


Figure 2.5: Screenshot of a chart generated by the Open XDMoD web portal showing CPU hours delivered over a two-year period broken down by job size (number of cores).

for commercial data centres but also for HPC data centres. Examples include batch scheduler controlled Dynamic Voltage and Frequency Scaling (DVFS) (see e.g., [Ge et al., 2005] and [Chung-hsing Hsu and Wu-chun Feng, 2005]), and simulation and optimization of HPC job allocation algorithms for jointly reducing communication and cooling costs (see e.g., [Meng et al., 2015] and [Kaplan et al., 2013]). While there is a wide array of research in the areas of data-centre building infrastructure, system hardware, system software, and applications, [Wilde et al., 2014] point out that current systems often miss proper instrumentation and, therefore, do not allow for the easy collection of required data. The Energy Efficient HPC working group [EEHPCWG, 2014] is trying to help the HPC community to define the needed HPC system and data centre instrumentation and provide guidelines on how to measure key energy efficiency metrics.

Application Support and Advice

One of the metrics the majority of HPC system operators are evaluated by is how well their platforms are utilized by applications. Typically, this is a maturity journey for platform operators, starting at optimizing for node utilization and moving over time into optimizing for scientific output. Dedicated application support and software analyst teams established by many of the large HPC centres are indicative of this. Traditionally, the support and advice model is mostly request-based, i.e., if an external user or developer experiences issues, for example with the performance of their parallel application, they contact the support function to ask for advice. In order to improve the user experience on HPC systems, it is beneficial to move from a pull-based support model (i.e., users contacting support if something goes wrong) to a proactive, push-based support model. In the push-based support model, users of an application are proactively notified if concerns with their applications arise. Next, we will look at *XALT*, and *TACC Stats*, two tools that aid the user and application support and advice.

XALT [Agrawal et al., 2014] is a mechanism for following users' jobs and environments on an HPC cluster. It provides a census of libraries and applications and automatically filters user issues, yielding exactly the type of job tracking information that most computing centres need or want. XALT is designed to track the execution information for applications that are compiled and executed on HPC clusters. XALT allows administrators and other support staff to consider demand when prioritizing what to install, support and maintain. Datasets, dashboards, and historical reports generated by XALT and the systems with which it interoperates will preserve institutional knowledge and lessons learned.

TACC Stats [Agrawal et al., 2014] collects data such as core-level CPU usage, socket-level memory usage, swapping and paging statistics, system load and process statistics, system and block device counters, interprocess communications, filesystems usage (NFS, Lustre ,Panasas), interconnect fabric traffic, and CPU counters and Uncore counters (e.g. counters from the Memory Controller, Cache and NUMA Coherence Agents, Power Control Unit). TACC Stats also provides a set of analysis and reporting tools which analyse TACC Stats resource use data and report applications with low resource use efficiency or that appeared to experience software or hardware issues. TACC Stats is initialized at the beginning of a job and collects data at specified intervals during job execution and once more at the end of a job. The data collected can be used to automatically generate analyses and reports such as average cycles per instruction (CPI),average and peak memory use, average and peak memory bandwidth use, interconnect traffic, and more on each job and over sets of jobs grouped according to user, application,project number, and date. These reports enable systematic identification of jobs, applications, or specific implementations of applications (such as building on different MPI stacks) which could benefit from architectural adaptation and performance tuning. In addition, these analyses are used for catching and flagging user mistakes such as allocating multiple nodes to a single-node shared-memory parallelized application or diagnosing system issues such as hardware and file-system failures.

2.2.2 Adaptive Application Architectures

Adaptive application architectures can be found across all classes and types of HPC applications, from tightly coupled parallel codes to heterogeneous, multi-component workflows (e.g., [Atkinson et al., 2017]). They represent a class of architecture patterns that change an application's behaviour, composition, or interaction with its environment as a reaction to observed dynamic changes in the application itself (intrinsic) or its environment (extrinsic) based on one or more objective functions. Common classes of objective functions for adaptive applications architectures are error mitigation and throughput optimization. Error mitigation functions try to shield an application from unexpected events such as node failures, I/O degradation, but also intrinsic events such as data staging errors or algorithmic issues. An example for throughput optimization would be a function that increases the number of application tasks running concurrently on a node as a reaction to environment metrics such as CPU, memory, or I/O if they suggest underutilization. In this section, we look at two high-level adaptive architecture domains: application (re-)configuration and resource management and illustrate them with practical examples.

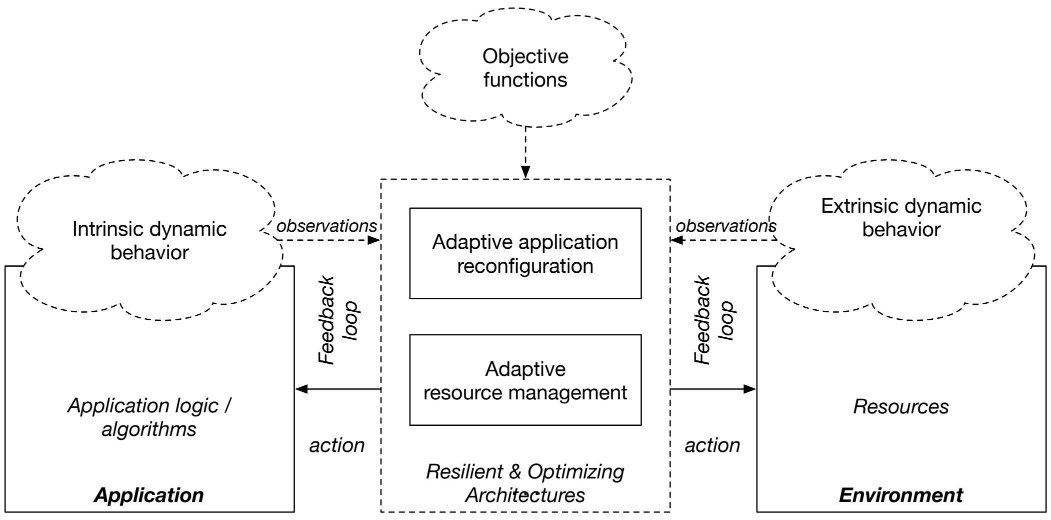

Objective functions
Intrinsic dynamic behavior
observations
Adaptive application reconfiguration
observations
Extrinsic dynamic behavior
Feedback loop
Adaptive resource management
Feedback loop
Application logic / algorithms
action
action
Resources
Resilient & Optimizing Architectures
Application
Environment

Figure 2.6: Adaptive application architecture patterns change an application's behaviour, composition, or interaction with its environment as a reaction to observed dynamic changes based on one or more objective functions.

Adaptive Application (Re-)configuration

We distinguish between adaptive application configuration and adaptive application re-configuration. The former allows an application to adapt to a new, unknown environment a priori, while the latter allows an application to change its configuration as it executes. Examples for an adaptive application configuration are the CoMPI implementation [Filgueira et al., 2011], which implements adaptive runtime compression of MPI messages and PRO-MPI [Venkata et al., 2009], which uses profiles of past application communication characteristics to dynamically reconfigure MPI protocol choices. In both examples, platform telemetry is collected and used to understand and choose configuration options for the application. A platform that would provide the relevant historic and real-time telemetry data to the applications would in both cases simplify the application architecture and adaption to other HPC platforms and architectures.

Adaptive Resource Management

Adaptive resource management is a set of architectural patterns that allow resource allocations to be altered during the runtime of an application. Most existing HPC workload managers use a static, a priori performance model. Fluctuations in the performance metrics of a resource, e.g., disk or network I/O hotspots are not monitored or acted upon. While this works well with static and homogeneous workloads, it fails with dynamic applications. Adaptive load balancing is an architectural pattern in which an application can re-schedule its workload within a static HPC resource allocation. Just as adaptive (re-)configuration, application-level load balancing might be triggered based on observed extrinsic changes in the environment or intrinsic changes in the application. Other adaptive load-balancing approaches are not tied into a specific application architecture. I/O aware schedulers such as [Yang et al., 2013] and [Herbein et al., 2016] can control the status of jobs on the fly during execution based on run-time monitoring of system state and I/O activities. We will not focus on these even though they equally rely on telemetry. In this section, we discuss two examples of adaptive application-level load-balancing: Charm++, a parallel object-oriented programming language and runtime system, and our I/O-aware load-balancing application called RADICAL Pilot.

Charm++ [Kale and Krishnan, 1993] is a C++ based parallel, message-driven, object-oriented programming language developed at the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign. Charm++ programs are decomposed into a number of objects called *chares*. When a program invokes a method on an object, the Charm++ runtime system sends a message to the invoked object, which may reside on the local processor or a remote processor. This message then triggers the asynchronous execution of code within the chare. Chares are mapped to physical processors by the Charm++ adaptive runtime system. The mapping of chares to processors is transparent to the programmer, which allows the runtime system to dynamically change the assignment of chares to processors during program execution. This decoupling is the foundation for capabilities such as load balancing, fault tolerance, and the ability to dynamically shrink and expand the set of processors used by a Charm++ program. During the execution of the program, the runtime system collects workload information on each physical processor in the background, and when the program hands over the control to a load balancer, it uses this information to redistribute the workload, and migrate the parallel objects between the processors as necessary.

RADICAL Pilot [Merzky et al., 2015b] is an application-level scheduling system that was initially developed to circumvent the static constraints and granularity of HPC workload managers. It works by submitting a single *pilot-job* or placeholder job to an HPC workload manager and once the job becomes active, allow user applications to schedule their jobs which are then executed via the pilot-job system within the placeholder job. In the example shown in figure 2.7, RADICAL Pilot was used to schedule a bag of homogeneous I/O intensive single-core jobs with a benchmarked runtime of around 10 *minutes* within a larger placeholder job. However, the job throughput of the application did not match the expectation: while the average task runtime in this example had been benchmarked at around 10 *minutes*, outliers in a first run had a run-

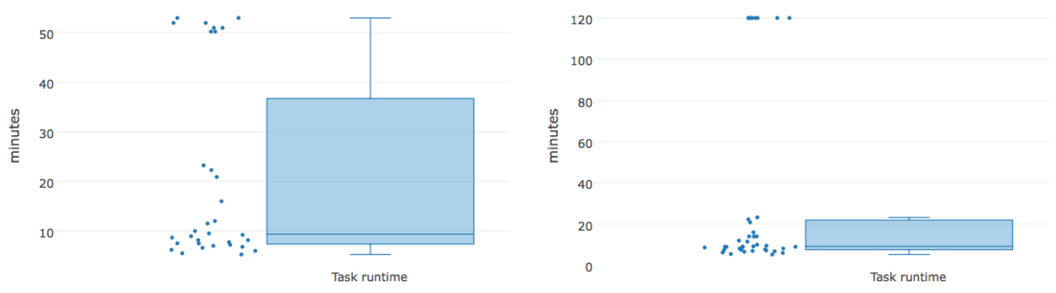


Figure 2.7: Box plot showing the runtimes of a set of homogeneous tasks. Extreme upper and lower values are due to I/O starvation on a subset of the executing compute nodes.

time of around 50 *minutes*, and in a second run, around 120 *minutes*. Each job had to stage in a data file of 10 GB before it could start executing. While this took less than a minute for the majority of jobs, it took an hour or more for the outliers in both cases. To understand this behaviour, the RADICAL Pilot node agents, the component responsible for application job execution on the individual nodes of the placeholder allocation were instrumented to collect CPU, memory, and Disk I/O telemetry. Instrumentation revealed that a specific subset of nodes exhibited a significantly degraded network filesystem I/O performance which caused the spike in data file staging time. While the cause could not be identified with certainty, the system instrumentation was used to continuously monitor node I/O performance and to use RADICAL Pilot's dynamic scheduling capabilities to move application jobs away from nodes that exhibit problematic behaviour and to remove the nodes from the internal resource pool.

While the Charm++ example uses adaptive load balancing to optimize for intrinsic dynamic behaviour, the RADICAL Pilot example uses adaptive load balancing to mitigate extrinsic dynamic anomalies. We have discussed hardware and software failure as one source of dynamic behaviour of the environment. Another source that is much more common, yet often more difficult to identify is application interference. Application interference and side effects are not unusual on multi-tenant HPC systems. [Dorier et al., 2014] for example have shown how the interference produced by multiple applications accessing a shared parallel file system concurrently becomes a major problem and often dramatically degrades I/O performance or even breaks data-intensive applications and, as a result, lower machine-wide efficiency.

2.2.3 Application Development

Analysis and optimization of HPC applications is a common and often repeated tasks for application developers. Analysis and optimization can be split into two categories: *1. Intrinsic* analysis and optimization of an application, i.e., its algorithms, data structures and communication patterns. *2. Extrinsic* analysis and optimization of an application that is executing on a specific HPC system, e.g., its compute and I/O performance and the resulting runtime characteristics. Intrinsic and extrinsic analysis and optimization are often interconnected processes. A common example is that extrinsically observed resource limitations on a specific platform might have direct impact on the intrinsic choice of algorithms or data structure of an application. In the opposite direction, an application's optimized communication pattern might impact which platform it can be mapped onto and how. This interconnected analysis and optimization process is carried out iteratively throughout the evolution and lifetime of an application, as user requirements change and applications migrate between multiple different HPC systems.

Besides these relatively long iteration cycles, dynamic and automated analysis and optimization which much shorter iteration cycles, such as autonomic computing, play an increasingly important role. Especially in scenarios where the intrinsic characteristics of an application can change during its runtime or where the target platform is not known before execution has started, analysis and optimization is ideally carried out in real time, without any user interaction, while an application is executing.

Even though we define intrinsic and intrinsic analysis and optimization as interconnected, their mechanics are not. Analysing application-specific algorithms, data-structures and communication patterns requires very different information, metrics and tools and has different objectives than the resource utilization of an application. While the former is often highly application (or application class) specific, we argue that the latter is not at all. Furthermore, the former is carried out entirely in the application domain, while the latter is carried out in the platform domain. From an engineering perspective, both should hence be independent entities and actors in a larger system with well-defined coupling points.

We argue that operating-system processes and their behaviour are the most common denominator for all applications and their extrinsic analysis and optimization. From a process perspective, it does not make a difference, whether the application is a tightly-coupled GPU-accelerated MPI application or a group of uncoupled, single-threaded Python scripts. Every resource interaction, whether it is computation, memory access, filesystem I/O, or network communication, are carried out and are observable through the operating-system process abstraction and its interface. As operating-system processes are under the control of the HPC system, it is intuitive that the platform provides an interface through which the key runtime characteristics of an application's processes are exposed and can be accessed by the user or by the application itself. Furthermore, it would be desirable that this interface is identical, or at least similar, between HPC systems.

However, none of today's HPC systems provide such interfaces or services. The result is that applications and users that require application process data for extrinsic analysis and optimization tend to develop hand-crafted, application specific solutions. These solutions are either based on instrumented application code or special-purpose processes (“monitoring jobs”) that are executed alongside the actual application. Process data is collected through the process control interfaces that the operating-system provides. However, the information gathered is confined to the operating-system (usually a single node of an HPC system) and still needs to be put into the larger context of the platform architecture and topology to understand the overall runtime profile of the application.

Interacting with low-level operating-system primitives appears to be asymmetric: while from a user’s perspective the application is submitted to, executed on and controlled by the HPC system software (queuing system, job manager, etc), further information about this process has to be extracted and pieced together from lower level entities. This methodology seems unnecessarily inefficient as it clutters application logic and creates potentially redundant code throughout many HPC applications. Furthermore, it is only accessible to users who have an in-depth technical understanding of the HPC system architecture and operating-system interfaces.

The application development lifecycle is an iterative process that takes an application from the initial concept to its implementations and refinements. In this process, an HPC application goes through many implementations, test, measure, and refinement iterations through which it slowly matures. Two aspects of the development workflow heavily rely on telemetry: test and measure, and debugging and diagnostics. Test and measure steps are conducted to understand how, for example a new feature or algorithm, behaves in different scenarios, at different scales, or on different hardware architectures. For this, a developer typically conducts a series of experiments or “test runs” and records the behaviour of the application. Some high-level results can be observed on the application-level, for example runtime and workload throughput, and do not require further insights into the application’s behaviour on the HPC system. But to gain a better understanding of a specific behaviour or in order to pinpoint an unexpected result, telemetry is necessary to correlate the application’s or algorithm’s behaviour and the behaviour of the operating-system processes representing the application. It might, for example, be important to understand memory consumption or data I/O patterns on the system to find and mitigate bottlenecks in an otherwise conceptually sound algorithm.

Another reason why thorough testing and measuring relies on telemetry data is system noise. When applications are tested at scale, this often happens outside “sterile” lab environments on large multi-tenant HPC clusters. Noise is inevitable on these systems, for example the disk I/O or network performance might be degraded or “jittery” on a subset of nodes or cluster partition due to a data-intensive application running in the vicinity. It is important for the developer to be aware of system noise so he or she can account for it and interpret application measurement results correctly.

Currently, there are no comprehensive and generally available software packages that aid the developer in extracting the relevant telemetry form HPC systems. Especially in the parallel programming world, instrumentation and performance tuning are essential in the development process and a series of tools exist for this purpose. However, these tools are generally narrow in scope and can only be used for parallel applications that fall into this narrow framework. Telemetry is extracted and processed internally, often with a focus on very low-level network communication metrics

The rest of the HPC application landscape is essential left alone with hand-crafted, often application-specific solutions for collecting and analysing telemetry data. In many cases, this takes up a significant amount of time and effort of the application developers, time that could better be spend elsewhere. The test and measure steps in the application development workflow can take disproportionally long because telemetry, the foundation for performance engineering and improvement is simply not available to the developers. Developers often add telemetry collection logic to the application code base, bloating it unnecessarily, and, in the worst case, creating side effects that distort the results.

Performance Analysis and Profiling

The traditional way of conducting performance analysis and tuning for high performance computing application has been an off-line approach with strong involvement from the user. A variety of performance measurement, analysis, and visualization tools have been created to help developers to tune and optimize their applications. These tools range from source code profilers such as ompP [Fürlinger and Gerndt, 2005], to communication and memory tracers such as PSINS [Tikir et al., 2009]. These performance tools typically rely on a five-phase workflow [Wagner et al., 2017], which consists of:

1. **Measurement**: Collecting a representative set of measurements, e.g., with increasing core counts for a scalability-focused analysis. A set of measurements allows us to better understand the evolution of key performance metrics and distinguish between general behaviour and behaviour specific for a certain number of cores or a certain input.

2. **Focus of analysis**: Getting an initial overview of the application behaviour, and detecting the overall structure. Based on this, selecting the focus of analysis. This allows developers to narrow down further analysis, make it more comparable between the different measurements of the set by removing, e.g., constant initialization time, and reducing the overall analysis effort.

3. **Performance modelling**: Using a performance model to determine the performance, efficiency, and evolution of key performance indicators.

4. **Detailed analysis**: Focusing and prioritizing the detailed analysis based on the outcome of the performance model. Gradually applying more advanced analysis techniques to understand the root causes of performance issues.

5. **Reporting**: Recording the performance overview, analysis results and recommendations and reporting them. This allows for the key elements of the performance analysis to be accessed by other users or analysts, and to be utilized for future analyses.

The basic purpose of application performance tools, is to help the user identify whether their application is running efficiently on the computing resources available. To do this, most performance tools offer instrumentation, measurement and presentation components for use in the five-phase cycle, and a few tools have begun offering an analysis component to better assist users. Notable work in the area of performance analysis tools include Paradyn [Miller et al., 1995], developed at the University of Wisconsin, and KOJAK [Mohr and Wolf, 2003], developed at the Research Center Jülich.

2.3 Current Challenges

We have identified five overarching challenges that HPC applications face when they want to utilize HPC telemetry as part of their architecture or workflows. While point solutions do exist for some of these challenges, none of them has been addressed holistically and in an application- or platform-agnostic way. Providing solutions for these five challenges that cater for a broad spectrum of users and use cases is the primary motivation for this research. We experience and identified these challenges throughout our own work on adaptive and distributed applications [3] as well as through analysing a cross-section of architectures of HPC applications that make use of telemetry. Furthermore, we have conducted a survey across multiple HPC centres to gain further insights into telemetry usage and support [4]. In summary, the five main challenges we have identified are:

Challenge 1: Improve Availability

While telemetry is usually collected by monitoring systems and scheduling systems on a continuous basis to support the HPC system operation teams, the data is usually not made available to a broader audience such as application developers and running applications.

Challenge 2: Deliver Accessibility

Telemetry is often not available in machine-readable formats or the formats diverge between different platforms and services. The same holds true for the interfaces exposing telemetry data which can range from API endpoints to flat-file downloads from a website.

Challenge 3: Facilitate Integration

For many use cases, telemetry or insights derived from telemetry analysis need to be integrated back into applications and services as input for optimization and resilience functions. No common approaches and patterns exist for this integration.

Challenge 4: Standardize Structure and Semantics

Telemetry is often difficult if not impossible to interpret without having an implicit understanding of the semantic context and the structure of the HPC system and application that was generating it. Little effort has been put into defining a framework for this. As a result, telemetry data often becomes useless outside the context it was originally collected in.

[3]See for example the development work presented in, [Weidner et al., 2017], [Weidner et al., 2016a], [Radak et al., 2013a], and [Jha et al., 2007a].

[4]Unfortunately, the number of responses was rather small. The survey design and results can be found in appendix A.

Challenge 5: Support Processing and Analysis

Telemetry data volumes can quickly become very large which requires advanced technology and infrastructure for processing and analysis. This can rarely be done within the scope of an HPC application itself. For example, trying to find suspicious I/O patterns in an application running across 10,000 processes is not a trivial endeavour. However, while they are well-developed for other application domains, facilities for analysing telemetry at scale and in real-time are not available to application users and developers.

2.4 Summary

In this chapter we have looked at the state of telemetry usage and management in HPC. We have identified the main sources of telemetry data across systems and applications and the approaches and systems that are in use today to extract and collect it. We have concluded that telemetry is predominantly used in HPC monitoring systems, extracted and managed via standard monitoring tools, and in a few advanced HPC application architectures, almost exclusively extracted and managed via one-off, custom-built application-specific solutions. Based on these insights and drawing upon related work, we identify five major challenges in the field of HPC telemetry: accessibility, availability, integration, structure and semantics, and processing and analysis.

In the next chapter, we will introduce the telemetry platform concept, show how it can address the challenges identified in this chapter, and provide new opportunities for HPC application research, architecture, and development.

Chapter 3

Telemetry Platform

In the previous chapter, we have looked at the broad spectrum of application areas for HPC telemetry and the current challenges that users and application developers are experiencing. In this chapter, we introduce the data platform paradigm and its application to HPC. We begin the chapter with an introduction to the data platform paradigm, an approach and conceptual architecture that has been shown to enable the efficient management and exploitation of large volumes of continuously generated data across many industry and research applications. We then set the scene for the main hypothesis of this thesis and discuss how the data platform paradigm can be applied to HPC telemetry and the opportunities that emerge from this in the areas of HPC systems research, decoupled and reusable architectures, resiliency and optimization, and application and service architectures that build on machine learning techniques.

In the second half of this chapter, we conduct a more detailed requirement analysis for a telemetry platform and present a conceptual design that covers usage modes, data models, interfaces and integration with HPC platforms. The conceptual design serves as the base for the prototype implementation we present in chapter 5. We deliberately decouple the conceptual design of a telemetry platform from its implementation, as the latter can potentially change drastically with the implementation context while the former will not. Lastly, we elaborate on related work in telemetry management systems and data models and how they relate to and complement the work presented in this thesis.

3.1 Data Platforms

Data analysis has undergone a radical change in the last decade with the advent of *Big Data* as a new paradigm in scientific research [Hey et al., 2009] and the industry alike. Collecting and storing vast volumes of data with the basic premise that it contains potential value has sparked a plethora of new and improved data processing and analysis techniques to sift through historic and live data to extract value. These techniques range from new programming models like MapReduce on elastic computational platforms [Dean and Ghemawat, 2010] to advancements in and wide adoption of time series analysis and machine learning. Many commercial systems and open-source building blocks exist that combine these techniques into a coherent technical platform, often referred to as *data platforms*. The majority of today's data platforms are made available via a Platform-as-a-Service (PaaS) model, either provided by an organization's internal IT department or cloud-hosted by a third-party vendor. PaaS service is a category of computing services that provides a platform allowing users to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. The original intent of PaaS was to simplify the application development process, with the infrastructure and operations handled by the PaaS provider, saving developers from the complexities of the infrastructure side (setting up, configuring and managing elements such as servers and databases). PaaS can improve the speed of developing an application, and allow the developers to focus on the application itself. With PaaS, the developer manages applications and data, while the PaaS provider manages runtime, middleware, operating systems, virtualization, servers, storage and networking. Data-Platform-as-a-Service (DPaaS) are characterized by predefined processes for data ingestion, storage, discovery, and analysis that are aggregated and exposed via APIs. Interaction with DPaaS takes place exclusively via these APIs instead of via the individual interfaces of the components comprising the platform. The constraints of pre-defined processes make the DPaaS approach difficult to realize for data platforms with a broad spectrum of different data, use cases, and integrations. It works well however for narrow use cases and well-defined data.

3.1.1 Data Science and Analysis Workflow

Data platforms are the key enabler for data science and analysis at scale. Without them, the overhead of data management, i.e., data ingestion and storage, and implementing

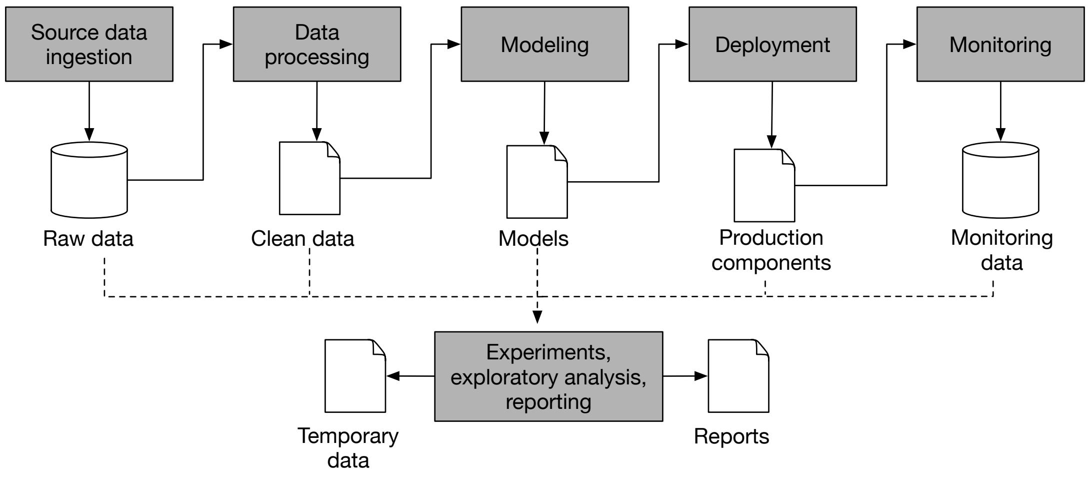


Figure 3.1: A high-level data science and analysis workflow. The boxes denote the key processes while the icons below are the respective inputs and outputs.

data processing at scale would increase the complexity and overhead of data science and analysis workflows by an order of magnitude. Data science describes the various scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data. Although data science objectives can range widely in terms of their aims, scale, and technologies used, at a higher level of abstraction most of them can be implemented as the workflow depicted in figure 3.1. The boxes denote the key processes while the icons below are the respective inputs and outputs. Depending on the objective, the focus may be on one process or another. Some of them can be rather complex while others trivial or missing. The details of the six steps are described below. Each step is supported by one or more data platform capabilities:

1. **Source data ingestion** is concerned with collecting relevant raw data from various source systems. This data can be file-based, extracted from an API, or consumed from a data stream. Data-platforms provide different tooling to support and automate data ingestion, such as data scheduling and pipeline services. The ingested raw data is stored in a part of the data platform often called the *data lake* (see e.g., [Miloslavskaya and Tolstoy, 2016]).

2. **Data processing** is concerned with turning the ingested source data into a "clean" form, suitable for use in the subsequent modelling stage. Data processing can involve changing the format of the data, applying a specific data model or schema to the data, and filtering out or interpolating incomplete records. A plethora of

tools exist in the data processing space, many of them based on parallel data processing frameworks, such as Apache Spark [Zaharia et al., 2016].

3. **Modelling** is concerned with using the data to build a formal model that describes the problem or question a data scientist is trying to solve. A model can be many things, from simple statistical models to more complex supervised and unsupervised machine learning models (see e.g., [Davison, 2003] and [Baltrušaitis et al., 2018]).

4. **Experimentation** is closely related to the modelling step. Modelling and experimentation is an iterative activity to validate and improve the accuracy or performance of the models. Data-platforms usually provide the capabilities to build and execute models efficiently and at scale. Machine learning frameworks like Tensorflow [Abadi et al., 2016] for example can execute model training in parallel, taking advantage of the underlying distributed data and processing architecture of a data platform.

5. **Deployment** is concerned with provisioning matured models in a production environment, so they can be used for continuous analysis, forecasting, and prediction. While not a core data platform capability, some data platform provide integrated tools and services for model deployment based on container technologies such as Docker and Kubernetes (see e.g., [Bernstein, 2014]).

6. **Monitoring** is concerned with observing key parameters and performance of the deployed models, i.e., ensuring the model is doing what we expect it to in production. This information is fed back into the model development process for further optimization. Common monitoring tools and frameworks, such as Grafana [GrafanaLabs, 2020] are provided as a capability by some data platforms to support this.

After this very brief introduction to data platforms, data-science and analysis workflows, and the relation between the two, we will now discuss how these concepts can be applied to HPC telemetry management.

3.1.2 Application to HPC Telemetry

The main hypothesis of our work is that by applying the data platform paradigm and architecture to HPC telemetry, we can at least partially address the main challenges that we have outlined in section 2.3. The rationale is that many of the telemetry-driven use cases and architectures in HPC closely resemble those we see in other areas of “big data”, especially in the areas of IoT and large-scale time seriesanalysis. The workflows of application developers and researchers are very similar to the high-level data-science and analysis workflow outlined in the previous section. Application developers equip their software with a multitude of sensors that continuously measure application-specific telemetry as well as telemetry that provides insights into the environment in which the application is operating. These can be operating-system or network metrics, metrics extracted from the platform services, such as workload managers, or metrics describing the progress of the simulation or computation, the application is carrying out. The insights extracted from the collected telemetry is then used to implement direct (autonomous) or indirect (human-in-the-loop) feedback loops to control the application while it executes (online) and to analyse its behaviour and interaction with its environment after the execution has finished (offline). With the steady increase of HPC application and system size and complexity, the volume of telemetry data generated is approaching a level that requires efficient and scalable infrastructure for storage, processing, and analysis. Data-platforms can provide this, and, if well integrated with the rest of the HPC ecosystem, become a commodity capability on which more efficient research and development workflows and application architectures can build.

Applying the data platform paradigm to HPC telemetry is conceptually quite simple: a scalable data storage and processing platform that is optimized for handling large-scale time seriesdata is built and integrated into an HPC environment. This telemetry platform provides the capabilities required to support the six steps of the data science and analysis workflow described in the previous section. The platform can either be hosted on-premise or public cloud based, important is that it is accessible transparently to the HPC users. The monitoring systems of the HPC platform are configured so that they continuously deliver telemetry data to the telemetry platform where it is stored within a semantic data model that reflects the structure of the HPC platform and its applications (“digital twin”). Ad-hoc analysis of telemetry data, e.g., for research or application development can be carried out using the integrated data analysis tools of the telemetry platform without having to transfer the data out first. Similarly, applications that use telemetry-driven feedback-control-loops to implemented resiliency and optimization strategies can use the telemetry platform to execute online and offline data processing and analysis tasks and provide the results back to the executing applications via the interfaces that integrates the telemetry platform with the HPC cluster.

3.1.3 Requirements

With the high-level concept of an HPC telemetry platform defined and based on the application areas and challenges identified in chapter 2, we can now define a number of functional requirements. These requirements inform the conceptual design of telemetry platforms. We distinguish between two types of requirements:

- must have requirements (R.F1 — R.F8) describe the basic capabilities for a telemetry platform to cater to the most common use cases and to address the gaps in existing solutions.
- should have requirements (R.F9 — R.F14) describe advanced capabilities that support additional and novel use cases that go beyond basic capabilities.

The majority of requirements (R.F1 — R.F14) are defined from the perspective of application developers and HPC researchers in an academic context. We made this choice deliberately, as we want to focus on this specific, yet large group of users and on the benefits and opportunities that comprehensive access to HPC telemetry gives them. We are aware that some requirements, such as *R.F2 — Open Access*, *R.F11 — Shared Solutions*, and *R.F12 — Public Access* are not generally applicable. Especially in the commercial HPC space and in environments in which competitive, sensitive or confidential research is carried out, these requirements can not be aligned with the privacy and security requirements of the users. Even in more open environments, these requirements can be challenging to satisfy. We pick this up in section 5.2.2 where we discuss the privacy and security concerns that arise from a fully open platform and how these can be mitigated.

Another perspective that is not represented comprehensively in the requirements below is the perspective of the HPC resource providers and operators. For this group of stakeholders, the potential impact on platform stability and performance, as well as again, security and privacy are a big concern. Requirement *R.F13 Usage-Based Billing Model* is currently the only requirement that comes directly from a platform operator's perspective and is concerned with the economic impact of a telemetry platform. We provide some additional details on cost vs. benefit in section 5.5.

In summary, the requirements below should be read as a set of *enabling* requirements, i.e., the requirements that unlock new value for users and researchers. This is what this work is focusing on. As an area of future work, we propose to look into the broader requirements of other stakeholder groups, i.e., a more comprehensive and formal requirement analysis, and to identify potential areas of conflict and their resolutions.

R.F1 Coherent Programmatic Access

A telemetry platform must allow users to access all of its capabilities and data via well-defined, coherent and machine-readable interfaces.

One of the main use cases for a telemetry platform is the direct integration of telemetry data by applications. In order to achieve this efficiently, all telemetry data must be available via a well-defined API that provides coherent syntax and semantics across all data sources. Similar to data platforms, telemetry platforms provide additional services that help developers to automate data processing, modelling, and analysis of telemetry (see Figure 3.1). In order to integrate these services seamlessly into HPC applications, these services must also be available through well-defined interfaces.

R.F2 Open Access

A telemetry platform must provide the users with open access to all application and system telemetry.

In order to enable research and novel telemetry-based services, telemetry data must be *democratized*, i.e., all data must be available to the entire user and developer community independently of the source or the producer.

R.F3 Interactive Analysis

A telemetry platform must provide user interfaces to explore and analyse telemetry data interactively

Before telemetry-based services can be automated and integrated into HPC applications, developers often need to experiment with telemetry data in order to explore and validate a specific algorithm or machine learning model. To enable efficient experimentation, a telemetry platform must provide *data-science workbenches*, interactive, graphical user interfaces with direct access to telemetry data.

R.F4 Coherent Semantic Data Model

A telemetry platform must organize telemetry data in a coherent semantic data model that captures the time-variant structure, properties and state of the HPC system and its applications.

To enable reproducibility, semantic interpretability, and cross-correlation of intra- and inter-platform system and application telemetry, a coherent, semantic data model is required that organizes telemetry data within a common framework of reference.

R.F5 Mutable Data Model

A telemetry platform must allow for the data model to be mutable, i.e., changeable and extensible.

To accommodate changes in HPC systems and application architectures, the semantic data model cannot be static but must permit changes and additions in structure, properties, and state.

R.F6 Variable Data Granularity

A telemetry platform must allow users to control the granularity and sampling frequency of data based on their requirements.

Requirements determining the granularity of data points in a telemetry dataset is dependent on the use case. For example, an application that uses telemetry to make long-term scheduling decisions requires a lot less granularity compared to an application that analyses MPI communication profiles. In order to cater to a broad spectrum of use cases, a telemetry platform should support variable, application-controllable data granularity.

R.F7 Customized Metrics

A telemetry platform must allow users to collect their own customized metrics and embed them in the coherent semantic model.

While a comprehensive set of common metrics, embedded in a coherent semantic model should be at the core of any telemetry platform, customized metrics still play an important role for many use cases. Especially correlating system behaviour with application logic requires the collection of customized application telemetry. A telemetry platform should enable the collection of customized system and application telemetry and interrelate them with the coherent semantic data model (see R.F4 Coherent Semantic Data Model) so they can be explored and understood outside the specific use case.

R.F8 Long-Term Persistency

A telemetry platform must provide long-term data storage capabilities to the users.

The availability of historic telemetry is important for several use cases, including comparative studies, long-term utilization analysis, and the application of machine learning techniques (see section 3.2). A telemetry platform should provide mechanisms to store and archive historic telemetry to the extent it is economically feasible.

R.F9 Scalable Data Processing

A telemetry platform must provide the users with the right tools to process and analyse telemetry data directly on the platform.

Moving data from its storage location to a system where it can be analysed can be time and resource consuming. Similar to data platforms, a telemetry platform must provide users with an environment in which they can directly process and analyse telemetry data at scale without having to move it out of the platform.

R.F10 Service Hosting

A telemetry platform should allow users to host their own application support services.

In order to reduce the complexity in HPC application design, core application capabilities and auxiliary capabilities are often decoupled into separate entities (see figure 3.2) with a well-defined functional scope. While core application logic has to run on the HPC cluster itself, auxiliary capabilities do not necessarily have to. In the case of telemetry-based capabilities, it often makes more sense to run the capability where the telemetry data and processing infrastructure resides, i.e., on the telemetry platform. In order for application developers to compose the services that comprise their application, a telemetry platform should provide a convenient way to host telemetry-based support services. For example, an I/O congestion detector that continuously analyses hardware telemetry during the runtime of an application and emits warnings via Representational state transfer (REST) API could be configured to automatically start up on a telemetry platform's hosting service and expose its interface to the core application component running on the HPC cluster.

R.F11 Shared Solutions

A telemetry platform should allow users to share their solutions with other users of the platform.

Many resilience and optimization capabilities can be built in a generic way so that they can become usable in more than one application. Reusability is an important strategy to reduce redundancy across applications. A telemetry platform can enable the development of reusable components through well-defined and coherent interfaces to telemetry (see R.F1 Coherent Programmatic Access). To further the adoption of reusable component, a telemetry platform should provide functionality for developers to share their solution with the rest of the HPC ecosystem, for example through a solution marketplace approach. A solution marketplace would allow the application developers to browse and search shared solutions and instantiate them to add a new capability, for example, a real-time I/O congestion detector, to their own application.

R.F12 Public Access

A telemetry platform should allow users to share their analysis and the underlying data with other users of the platform as well as to disseminate it publicly.

In order to build more confidence and enable reproducibility in systems research, scientists need a way to publish and share large data sets that comprise the data points used for the aggregated results published in a piece of scientific research. So far, data, if shared at all, is shared outside the context in which an experiment was conducted. Data is uploaded to university file servers or public file-sharing services. If the data could be published and shared directly on a telemetry platform, peers could review the data where it was produced using the provided tools and even correlate it with other telemetry data that was collected at the same time of the experiment or any other time.

This would be a step-change in research data management and would build additional trust and rigour in HPC system research.

R.F13 Usage-Based Billing Model

Telemetry platform usage should integrate with the usage-based billing models of the HPC systems it serves.

Building and continuously operating a telemetry platform can generate significant costs for a platform provider. Depending on the use case an HPC application might use an extraordinarily high amount of telemetry platform resources, for example, due to high data density and velocity combined with processing-intensive analytical pipelines, or no resources at all. A billing model that reflects the actual resource usage would benefit both, the platform providers and the users.

R.F14 Implementation-Independent Specification

The telemetry platform, its interfaces, data model and capabilities must be described independently of its implementation.

In order to allow for broad adoption of the telemetry platform paradigm, it is important to keep the specification of the telemetry platform and the implementation separate. HPC platforms come in many variants, and as a result, architectures and software stacks can vary significantly. A telemetry platform should hence be described in terms of its capabilities and interfaces. This will allow HPC centres and commercial vendors to implement and integrate the platform according to their specific context, while the applications and use cases remain portable between different implementations.

3.2 Opportunities

Introducing telemetry as an HPC system service can substantially change the way HPC software is written and how applications are developed and run. It can also have a positive impact on how HPC research is conducted and disseminated. The five opportunities we introduce in this section are the main drivers for our work.

3.2.1 HPC Systems Research

Computer science research into HPC systems and their supporting systems and services, such as distributed filesystems or runtime systems, is one of the key drivers for HPC system evolution. Virtually all experiments carried out on HPC systems need to collect vast amounts of data that are then interpreted and published as part of the scientific process. A unified telemetry management system can provide the framework and building blocks for this scientific apparatus. Not only the collection of data but also the long-term, immutable storage and cataloguing of results can streamline the experimental workflow and increased the confidence in the results reported.

Especially reproducibility is still a problem in the community. A survey conducted at the 21st International European Conference on Parallel and Distributed Computing [Hunold, 2015] revealed that the majority of the participants believe that the state of reproducibility needs to be improved in the domain of parallel and high-performance computing and that the majority of the results presented in papers that they receive for review are unlikely to be reproducible. While a telemetry management system alone will not solve this issue, it can still contribute to the solution through:

(i) **Providing research-friendly environments:** The development and testing of new and experimental HPC *system* features and service can not be conducted on live HPC systems. While the damage caused by unexpected application behaviour is usually limited to the user's context, faulty system services can potentially impact the productivity of an entire HPC user community. For this reason, most systems research has to be conducted in isolated sandbox environments. This presents the keen systems researcher with a problem: the isolated environments that they need to conduct exploratory research can be very difficult, complex and costly to build, set up, evolve and maintain. It can require everything from setting up hardware, to the orchestrated management of operating systems, shared file systems and HPC services. Unless a shared testbed environment is jointly managed and operated within a larger research group, it is very difficult for the individual researcher to embark on such an effort alone.

(ii) **Enabling reproducible and comparable results:** The second problem that researchers and experimental systems engineers are confronted with, is to ensure the reproducibility, preservability and documentation of their experimental environments. Good research practice demands that published results can be reproduced and build upon by peers. In the case of HPC systems research, this might require the reproduction of an entire sandbox environment to use the tools and apply the methods that were used in the original work. That this process is still far away from being trivial and widely adopted was shown by a 2015 study [Ivie and Thain, 2018] conducted across 400 ACM conference and journal papers. The results of this study showed that only 85 papers (21.2%) provided links to their codes. The study showed further that only the codes of 32.3% of the papers could be recreated within 30 minutes, the codes of another 16% of the papers could be rebuilt with extra effort, and that it was difficult or even impossible to rebuild the codes of the remaining 51.7% papers.

3.2.2 Decoupled Application Architectures

The main value proposition of the telemetry platform approach is that it efficiently organizes all services around telemetry management and usage in a platform and exposes its capabilities as a service. Conceptually, it proposes a separation of concern between telemetry management functionality and other, more application-specific functionality or domain logic. Figure 3.2 shows two different levels of decoupling: on the left-hand side, it shows a typical monolithic application architecture (as e.g. found in [Jha et al., 2007b]) where all the components required for telemetry management as well as resiliency and optimization functionality are part of the same application. This design has four significant drawbacks to consider:

(i) It is difficult to share the telemetry collected within the application context with other applications or users, and it requires extra effort to disseminate it as a supporting data asset for a publication.

(ii) A lot of application complexity is encoded outside the application core logic. This can bloat the application code significantly and add additional sources of failure that can lead to a decrease in overall application stability and reliability.

(iii) Application-specific telemetry management components are often developed for specific cluster architectures and system telemetry extraction points. This can make it difficult to "port" the application to other systems.

(iv) If embedded in an application monolith, resiliency and optimization logic often becomes difficult to reuse as it is usually tightly coupled with custom telemetry management and domain logic.

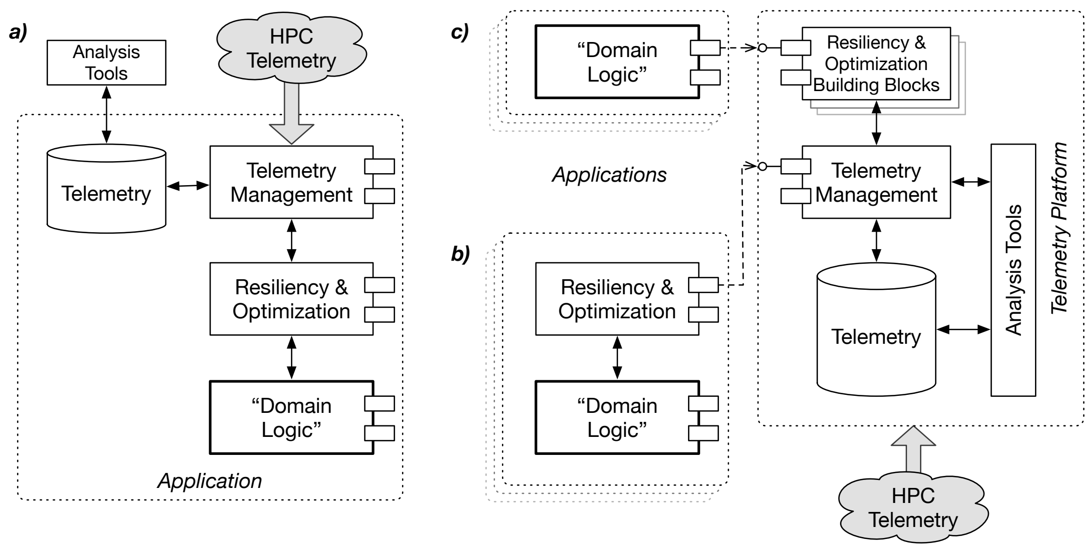


Figure 3.2: This component diagram shows how a telemetry platform can help to decouple monolithic application architectures (a) into more service-oriented architectures in which telemetry management is consumed as a service (b) and resiliency and optimization buildings blocks can be reused (c).

The right side of figure 3.2 shows how a telemetry platform can help to address these four issues by allowing more service-oriented application architectures in which telemetry management (b) and resiliency and optimization building blocks (c) can be integrated into an application design via loosely coupled capabilities. Especially since the development of telemetry management, resiliency and optimization capabilities are complex and do not contribute to the scientific advancement of an application itself, these capabilities are attractive candidates for consumption as a service as it will decrease effort and cognitive load on the application developer. Instead of explicitly postulating new, more decoupled application designs, a telemetry platform presents the opportunity to provide the right incentives for these architectures to evolve naturally.

3.2.3 Resiliency and Optimization

With the growing complexity and scale of HPC systems, application performance variation has become a significant challenge for efficient and resilient system management. Resilient execution covers a broad spectrum of methods and tools that are designed to prevent HPC applications from entering an unexpected, terminal state of failure. A terminal state of failure is a state in which the application has stopped execution before it has produced its expected results and from which it cannot recover. The causes can be manifold: application crash due to faulty platform hardware and application crash due to application (software) bugs, but also premature termination of the application by the scheduler can be listed. The reasons for the latter again are manifold and can range from applications exceeding their allocations (or wall clock time) to resource retention by the scheduler in favour of a more "urgent" application [Beckman et al., 2007].

Another consequence of the complexity is, that only large projects with enough resources and domain-specific expertise can afford to implement telemetry-based optimization and resilience patterns. For many smaller projects, this remains infeasible, which effectively creates a two-class system in which some applications become highly efficient HPC system tenants with a near-optimal return on investment (billed CPU hours), while others, the long tail, are often forced to remain in a suboptimal space. Supporting the long tail of applications by making telemetry more accessible will allow the HPC community to move from resilience and optimization point solutions and custom tooling to solutions that are much easier to implement, adapt, and share. If more applications can benefit from telemetry-driven optimization and resilience patterns, both, the efficiency and ultimately the scientific productivity of the individual applications can be increased. This would lead to an overall efficiency increase for HPC systems by reducing unproductive utilization of resources that are caused by hardware and software failures, human error, suboptimal resource usage, and inadequate adaption to new architectures. In conclusion, the key improvements that our HPC telemetry platform can bring to resilient and optimizing software architectures can be summarized as:

(i) **Lower adoption barrier:** without having to consider the inherent complexity of telemetry extraction, transport, storage, and analysis, we argue that the adoption of advanced resilient and optimizing application architectures becomes much easier.

(ii) **Optimized data handling:** with telemetry data handling managed centrally by a specialized system, common pitfalls and mistakes that can lead to unexpected and hard to diagnose side effects can be avoided effectively.

(iii) **Reduced redundancy:** with telemetry management services in place, redundant implementations of telemetry management capabilities at the application level can be reduced to a minimum. This reduces both development effort and time.

Together, these improvements help with a wider adoption of advanced architecture patterns that rely on telemetry, which would in turn contribute to an overall more stable and optimized HPC application landscape. This ultimately leads to increased scientific application throughput and better utilization of HPC resources. In the wake of emerging exascale systems and applications, an increased range of resilient and optimizing architectures becomes increasingly important.

3.2.4 Machine-Learning Approaches

With growing size and complexity of HPC systems, both the volume of data, and the variation of system and application behaviour and anomalies observed in that data will increase as well. Resiliency and optimization techniques, especially at scale, and when developed to support a broader set of applications and use-cases, will need to apply advanced data processing and analysis techniques to cope with the volume and variation. As an alternative to more explicit approaches, machine learning-based (ML) approaches are becoming increasingly popular and a novel body of work is emerging in the HPC literature that is concerned with applying ML algorithms to resiliency and optimization problems. Especially online-approaches [Fontenla-Romero et al., 2013], i.e., ML algorithms and architectures that can predict and classify on real-time data can help to alleviate systemic issues around premature job termination, reduced performance, and wasted HPC platform resources. [Tuncer et al., 2017a] for example compares ensemble learning techniques to classify different commonly observed anomaly types, such as memory leaks, CPU throttling due to thermal issues, and resource contention. In [Bhatele et al., 2015], the authors apply supervised learning algorithms, to perform regression analysis on network and communication telemetry in order to create models to predict the execution time of communication-heavy parallel applications. Another example can be found in [Kasick et al., 2010], which uses CPU instruction-pointer samples and function-call traces to identify issues with PVFS I/O node servers.

A critical problem in automated anomaly diagnosis based on application and system telemetry is the overwhelming volume of data collected and processed at runtime [Ibidunmoye et al., 2015]. While ML-based approaches can significantly alleviate the data processing pressure at runtime, it still requires data infrastructure support and support for large-scale data processing for offline model training. Out of the existing research we have surveyed, only a few provide any details of the development workflow and implementation architecture utilized. The ones that do, suggest that the telemetry data used for model training was manually transferred from the HPC platform to a separate system where the model training and analyses were conducted. Due to these data management and infrastructure challenges, the application of ML-based approaches remains difficult and mostly confined to research prototypes and proof-of-concepts. Our telemetry platform provides the foundation critical to wider adoption and application through:

- **Long-term data storage:** One key requirement for effectively applying ML-based techniques is the availability of "historic" data, e.g., telemetry data from previous HPC application runs for training of ML models. A telemetry platform provides these long-term telemetry data storage capabilities along with the facilities necessary to efficiently manage it.
- **Processing at scale:** The second challenge for ML-based techniques is preparing and processing telemetry data at scale as part of the model training workflow. Adopting the basic architecture properties of data platforms, a telemetry platform provides scalable compute capabilities and capacity colocated with the telemetry data. This makes complex and error-prone manual data transfers into and out of the HPC platform as well as the need to provision external data processing facilities redundant.

As one of the new and promising data-driven approaches in HPC application and system resilience and optimization, we use machine learning based application anomaly detection as the evaluation use-case for our telemetry platform prototype in chapter 6.

3.3 Related Work

The telemetry platform concept builds on the work that has been done in the area of HPC systems monitoring. The idea that systems monitoring approaches leave out the users of the system and that a subset of monitoring data in a format users can easily interpret and utilize has been discussed in [Moore et al., 2015]. As part of an initiative to open monitoring data to users at Los Alamos National Laboratory, the authors propose a concrete architecture combining a back-end data transport layer based on RabbitMQ [Videla and Williams, 2012] and a web-based front-end interface that provides telemetry access for users. While the motivation of their work is similar to ours, their approach does not consider applying a holistic data platform architecture but rather focuses on telemetry transport and presentation. A similar motivation and approach are presented in [Thaler et al., 2020]. Here the authors present a hybrid approach to HPC telemetry and hardware log analytics. Their approach focuses on data extraction and transport at scale. Similar to our approach, they use Apache Kafka [Kreps et al., 2011] as a data stream broker between the telemetry collection components (producer) and data analysis components (consumer). Neither of the two approaches explicitly discusses telemetry data models.

A closely related area of research and practice is data centre telemetry. Data centre telemetry platforms try to solve a related problem, namely the collection of data that is generated by the hardware, software, and applications in a data centre. Open standards exist for data centre telemetry, such as OpenTelemetry [Ferreira, 2021], a collection of tools, APIs, and SDKs to instrument, generate, collect, and export telemetry data. Commercial vendors of monitoring and telemetry platforms, such as Datadog [Datadog, 2021] and AWS CloudWatch [Services, 2021] support these open standards and provide telemetry collection and storage capabilities at extreme scales and can be provisioned and consumed as a (cloud) service. In terms of their design and capabilities, these standards and systems have the closest resemblance to what we envision as an HPC telemetry platform. It would be conceivable to build an HPC telemetry platform around these services and standards. The main differences between these and the approach we propose are twofold. Firstly, we propose an explicit telemetry data model that provides explicit insights into the "physical" structure of an HPC platform, its applications, and the relationship between the two. These structures are only implicitly discoverable in existing data centre telemetry approaches. Secondly, compared with existing data centre telemetry approaches, we propose a much closer integration between telemetry management, analysis, and application feedback loops into a single, coherent platform. While we don't consider it in this work, some aspects of OpenTelemetry, such as metric taxonomy and nomenclature could certainly be re-used or adapted for the data models and interfaces we propose.

Another category of related work worth mentioning here is *(Grid) Information Services*. They are of relevance to our work as they approach telemetry from a more user- and developer-centric viewpoint rather than from the viewpoint of a system operator. They provide particularly interesting insights, as many initiatives for HPC system information data standards and interfaces were rooted in, and driven by the distributed computing communities. Grids are a form of distributed computing where many networked, loosely coupled computers acting together to perform very large tasks. While many Grids in the early days consisted of very heterogeneous computing resources, from cluster nodes to idle workstations to home computers (volunteer computing), Grids have more and more merged into grids of HPC systems. Commonly used Grid systems include, Globus Toolkit [Foster, 2006], Condor [Thain et al., 2003], and the Advanced Resource Connector (ARC) [Ellert et al., 2007]. All Grid computing approaches have in common that they need to understand the state and dynamic properties of their distributed resources in order to make informed decisions about workload placement and to mitigate failures that are omnipresent in a dynamic, distributed environment. Pegasus [Deelman et al., 2015], a distributed workflow system, for example, relies on process-level telemetry to schedule workflow tasks across federated HPC infrastructure and to provide resilience and fault-tolerance. It uses an execution wrapper, which acts in between the remote scheduler and the executable, gathering telemetry about the executable run-time behaviour. Another example is our development, RADICAL Pilot [Merzky et al., 2015b], a distributed task scheduling framework, which also uses execution wrappers to collect telemetry, as well as to discover system properties like the number of cores and memory per node, that are then used to optimize the internal task scheduler for a particular system. All these systems expose some form of telemetry data model and interface to their users. Globus Toolkit for example provides the Metadata Service (MDS4) [Schopf et al., 2006a], and ARC provides the ARC Information System [Kónya and Johansson, 2010]. The Open Grid Forum (OGF)[1] has developed an open interface standard and data model for Grid telemetry. This standard is for example implemented in the Simple API for Grid Applications (SAGA) [Goodale et al., 2006] which provides the foundation for RADICAL Pilot. None of the systems and approaches however explicitly capture system and application structure and its temporal variance as we propose it with our telemetry graph approach.

This section provides only a small snapshot of related work. Other related work is

[1]`https://www.ogf.org/ogf`

mentioned wherever appropriate in the remaining chapters.

3.4 Summary

We have identified several opportunities in the areas of HPC application development and platform services that would benefit from a new, more structured approach to telemetry management, namely, improving system research workflows, increasing portability and reusability of applications and services, a wider proliferation of resilient and optimizing architectures, and enabling novel resiliency and optimization methods based on machine learning and artificial intelligence. These opportunities suggest that telemetry management needs to play a more central role in HPC system infrastructure and services as a lot of advanced use cases rely on the availability of and easy access to telemetry. A more overarching opportunity for rethinking telemetry management is to build the foundation for upcoming *exascale* systems [Allcock et al., 2011]. The inevitably increasing error rates of these systems as a result of scaling up, and complexity that is becoming increasingly difficult to grasp and manage by human actors put telemetry at the centre of the stage as the enabling technology for a new generation of smart adaptive and resilient systems and applications. Next, we will define a formal telemetry data model to address some of the requirements we have listed in this chapter.

Chapter 4

Telemetry Data Model

Our survey of related work, use cases and requirement analysis has provided us with clear indicators that one of the important missing capabilities in the existing HPC telemetry landscape is a data model that makes telemetry easy to organize and understand and that makes it universally interpretable and comparable. Consequently, a data model to organize telemetry must be at the centre of any telemetry platform. In this chapter, we introduce our telemetry data model called *telemetry graph*, which is based on a time-variant labelled multigraph to represent the evolving and changing structure of HPC system, applications, and the relationship between the two. We begin this chapter with the high-level design concepts of the telemetry data model and how the requirements defined in chapter 3 are reflected in them (4.1). In the next section (4.2), we provide a formal definition of the graph model, and in section 4.2.4 we describe how telemetry is embedded and organized within it. In section 4.3 we show by example how our abstract model definition can be instantiated for a common HPC system architecture. Lastly in section 4.4, we present a hybrid database design based on both a graph- and a time-series-database.

4.1 Design Concepts

The telemetry data model provides a structural and semantic framework for organizing telemetry data. It is a representation of concepts and the relationships, constraints, rules, and operations to specify telemetry data semantics. Our aim is not to introduce yet another platform- or application-specific model orthogonal to already existing approaches. Instead, driven by the challenges and requirements we have identified, we set out to develop a more generic, extensible data model that is platform and application agnostic and can incorporate existing telemetry data sources in a common context. In order to address current issues and to better support telemetry-driven use cases, we have defined a number of functional and non-functional requirements for a telemetry platform in chapter 3. While many of the requirements aim at telemetry platform capabilities, several requirements are directly relevant for the design of the telemetry data model. These are:

R.F4 Coherent Semantic Data Model: A telemetry platform must organize telemetry data in a coherent semantic data model that captures the time-variant structure, properties and state of the HPC system and its applications.

R.F5 Mutable Data Model: A telemetry platform must allow for the data model to be mutable, i.e., changeable and extensible.

R.F6 Variable Data Granularity: A telemetry platform must allow users to specify the granularity and sampling frequency of databased on their requirements.

R.F7 Customized Metrics: A telemetry platform must allow users to collect their own customized metrics and embed them in the coherent semantic model.

R.F14 Implementation-Independent Specification: The telemetry platform, its interfaces, data model and capabilities must be described independently of its implementation.

The first two require the data model to exhibit a flexible and extensible semantic structure, while the other two require the data model to accommodate flexible and extensible telemetry data. From these, we derive two main design concepts for the telemetry data model: (1) using graphs as semantic structure, and (2) a distinction between abstract and concrete model. Graphs provide an intuitive abstraction to capture the time-variant structure, properties and state of the HPC system and its applications, while at the same time allowing for the model to remain changeable and extensible. Distinguishing between an abstract model definition and concrete realizations allows us to apply the same concepts and structure to different platform and application models. In the following sections, we will discuss both aspects in more detail.

4.1.1 Graphs as Semantic Structure

Ideally telemetry is represented in a semantic context that represents the structure and properties of the HPC system and its applications at the time of its collection. This goes back to the idea of creating a “digital twin”, i.e., a digital replica of the elements and dynamics of HPC system and applications. In such a context, telemetry data becomes universally understandable and interpretable.

We call those structures and their interrelationship *anatomies*. Anatomies are modelled as time-variant graphs and provide the overarching structure for all telemetry in a telemetry platform. We distinguish between platform anatomy, which represents structure and properties of the HPC platform, and application anatomy, which represent the structure and properties of the applications running on it. Anatomies are composed of a fixed set of relevant component types, but the lifetime of these components and the relationships between them can evolve over time. Hence, we have a dynamic semantic structure. Platform anatomy for example, could be the intra- and inter-node hardware layout. Component types could for example be ”node”, ”CPU”, ”network interface”, and so on. An example for application component types could be ”operating-system processes” and ”threads”.

We call the mapping between the evolving platform and application anatomies *allocations*. An allocation can for example be the representation of a thread running on a specific CPU core of a node. Together, anatomies and allocations form a holistic semantic structure for platform and application telemetry.

4.1.2 Abstract and Concrete Model

In order to fulfil requirement *R.F14 Implementation-Independent Specification*, we distinguish between the abstract model definition and its concrete instantiations. The abstract model describes the rules on how the structure and semantics of an HPC system and application ecosystem can be realized. It does not make any specific assumptions about the architecture of HPC systems and applications. The only assumption it makes is that there is one type of structure describing a platform and another type of structure describing the applications and that these two structures interact with each other when the application is executed on the platform. A concrete model instantiates the abstract model for a specific context. This context is defined by a specific platform and application architecture and maps the abstract structural components to concrete ones, for example “nodes”, “CPUs”, and “processes”. We make this distinction to avoid defining a data model that is too narrow or opinionated. Splitting up the information into an abstract model and concrete realizations also allows us to evaluate the data platform concept with a simple model that is sufficient for this research without claiming to be generally applicable. Ideally, if the HPC communities adopt the data platform concept, a few common concrete models would emerge within those communities. With all the different concrete models adhering to a common abstract model, they can exist within the same data platform implementation.

4.2 Abstract Graph Model

We distinguish between an abstract model and concrete *realizations* of it. The abstract model, which is formally introduced in this section, describes the graph model and the relationship between vertices and edges without predefining the *vertex types* and *edge types*. These are only defined in a concrete realization of the model of which we give an example later in this chapter. We define the semantic structure of HPC system and applications formally as a labelled, directed multigraph, called the telemetry graph[1]. The basic definition of a *telemetry graph* is as follows:

$$
\begin{aligned}
TG = (V, E, s, t, \Sigma_V, \Sigma_E, \ell_V, \ell_E) &\text{ where} \\
V &\text{ is a set of vertices, and } E \text{ is a set of edges,} \\
s\colon E \to V &\text{ assigning to each edge its source vertex,} \\
t\colon E \to V &\text{ assigning to each edge its target vertex,} \\
\Sigma_V \text{ and } \Sigma_E &\text{ are finite alphabets of the available vertex and edge labels} \\
\ell_V\colon V \to \Sigma_V &\text{ a map describing the labelling of the vertices,} \\
\ell_E\colon E \to \Sigma_E &\text{ a map describing the labelling of the edges.}
\end{aligned}
\tag{4.1}
$$

[1]Multiple multigraph definitions exist in the literature. We follow the definition found in [Diestel, 2000] and [Bollobás, 2013] where multigraphs are allowed to have loops, i.e., edges that connect a vertex to itself. Other authors call these pseudo-graphs, reserving the term multigraph for the case with no loops.

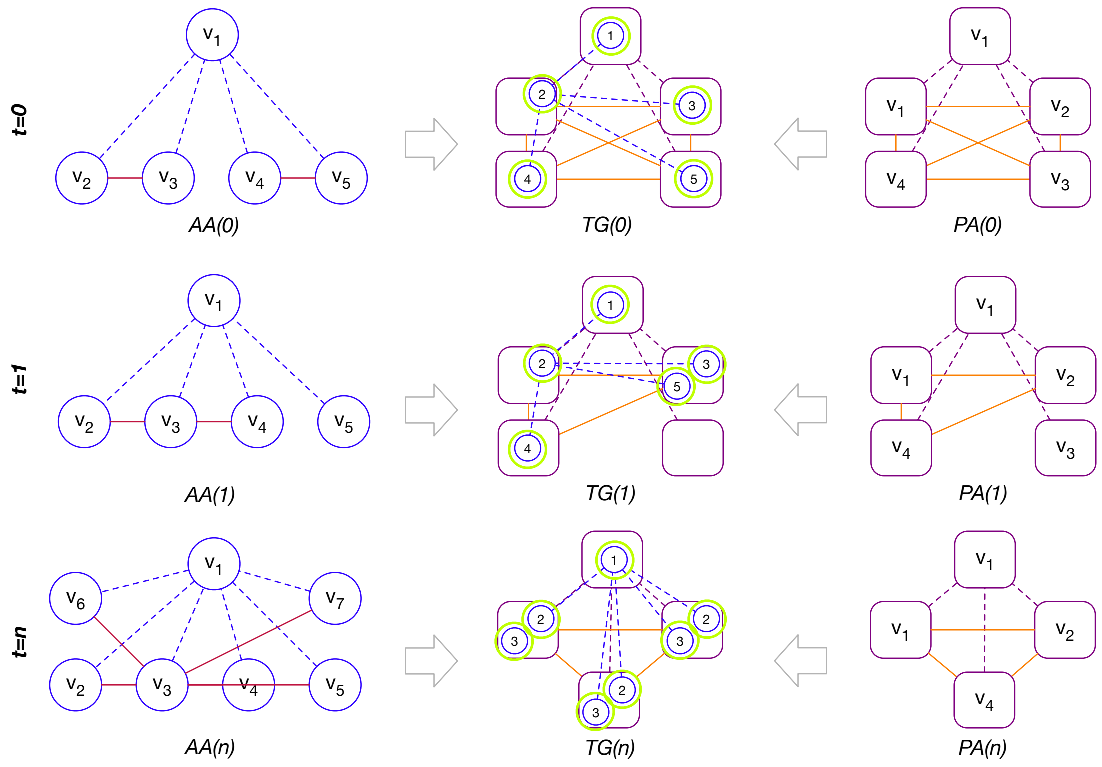


Figure 4.1: The telemetry graph (TG) consists of the time-variant application anatomy (AA) and platform anatomy (PA) graphs and the mapping between the two via allocation edges. Vertex and edge colours illustrate their types and are used throughout this chapter.

The rest of this section unfolds this definition further into the specific properties of a *telemetry graph*:

- *Platform anatomy* (PA) and *application anatomy* (AA), sub-graphs of TG, which present the platform and application components and their relationships.
- *Structure edges*, which capture the "physical" structure of platforms and applications.
- *Interaction edges*, which capture interactions between components.
- *Allocation edges*, which connect platform and application sub-graphs and inform the localities of application components within a platform.
- *Attributes*, which further qualify the labels (ℓ_V, ℓ_E) and distinguish between *Properties*, *Measurements*, and *Events*.

Figure 4.1 shows a simplified example of an abstract telemetry graph and its components. An application (left) is modelled as an application anatomy (AA) sub-graph, consisting of a number of vertices (blue ●), i.e., application components, (e.g., processes or threads) and a number of structure edges (blue ●) and interaction edges (purple ●). The HPC system (right) is modelled as a platform anatomy (PA) sub-graph, consisting of a number of vertices (violet ●), i.e., platform components (e.g., nodes, CPUs, or cores), and a number of structure edges (violet ●) and interaction edges (orange ●). The centre of the figure shows and overlay of both, (PA) and (AA) which represents allocation of application to platform components (green ●), i.e., the complete telemetry graph TG (AA interaction edges were omitted to increase legibility). The changing structure from top to bottom ($t = 0, ..., t = n$) shows the time-variance of all vertices and edges in a telemetry graph. In a concrete instantiation of this abstract telemetry graph, we could for example identify PA vertices as CPUs and AA vertices as processes or threads of an application. We discuss a concrete example of a model instantiation in section 4.3. The next section defines platform and application anatomy graphs in more detail.

4.2.1 Anatomy Sub-Graphs

Anatomy sub-graphs model the structure of HPC systems and applications. Consequently, we distinguish between *platform anatomy graphs* (PA_i) and *application anatomy graphs* (AA_j). Together, (PA_i) and (AA_j) comprise the complete telemetry graph (TG). Under the assumption that any HPC system will run more than one job, sequentially or in parallel, having multiple AA_j, i.e., $j > 1$ is intuitive. We also chose to allow multiple PAs, i.e., $i > 1$, to make the model applicable to federations of HPC systems in the future. All discussions and practical examples in this work assume a single platform, i.e., $i = 1$. To reflect the distinction between application and platform anatomy graphs, we expand the definitions of V and E in equation (4.1) with a distinct set of platform and application anatomy vertices and edges:

$$
\begin{aligned}
&V = (V_{PA\,i}, V_{AA\,j}) \text{ where} \\
&\quad V_{PA\,i} \text{ (●) is a set of platform, and } V_{AA\,j} \text{ (●) a set of application vertices.} \\
&E = (E_{PA\,i}, E_{AA\,j}) \text{ where} \\
&\quad E_{PA\,i} \text{ is a set of platform, and } E_{AA\,j} \text{ a set of application edges.}
\end{aligned} \tag{4.2}
$$

With this definition in place, we can now distinguish between vertices and edges that belong to an application's structure and vertices and edges that belong to an HPC system's structure. By "structure" we mean the representation of applications and platforms at the operating-system and hardware levels, i.e., their composition of (distributed) processes, threads, other operating-system primitives, nodes CPUs, cores, and so on. Next, we define two types of platform and application edges that will help us to model structure and interaction within the anatomy sub-graphs.

Structure Edges

Structure edges are one of two edge types in the telemetry sub-graphs. They represent the underlying logical or physical structure that organizes platform and application components. Structure edges can only be defined between vertices of the sub-graph-type, i.e., they can connect platform edges E_{PA} to platform edges and application edges E_{AA} to application edges. They cannot connect application edges to platform edges and vice versa. This is handled by allocation edges which are described in section 4.2.2. To reflect the distinction between application and platform structure, we expand the definitions of s and t in equation (4.1) with distinct mapping functions for platform and for application structure:

$$
\begin{aligned}
&s = (s_{StructAA}, s_{StructPA}) \text{ where} \\
&\quad s_{StructAA} : E_{AA} \rightarrow V_{AA} \text{ assigning each application structure edge (●) its source } V_{AA}, \\
&\quad s_{StructPA} : E_{PA} \rightarrow V_{PA} \text{ assigning each platform structure edge (●) its source } V_{PA}. \\
&t = (t_{StructAA}, t_{StructPA}) \text{ where} \\
&\quad t_{StructAA} : E_{AA} \rightarrow V_{AA} \text{ assigning each application structure edge (●) its target } V_{AA}, \\
&\quad t_{StructPA} : E_{PA} \rightarrow V_{PA} \text{ assigning each platform structure edge (●) its target } V_{PA}.
\end{aligned} \tag{4.3}
$$

With structure edges in place, we can now create a telemetry graph that can represent the structure of an HPC system and its applications as individual sub-graphs. In order to model the interaction between individual platform and application components, we define interaction edges next.

Interaction Edges

Interaction edges are used to model interactions between components. Interaction edges can be defined (1) between vertices of the same platform anatomy sub-graph, (2) between the vertices of different platform anatomy sub-graphs (in federated systems), (3) between vertices of the same application anatomy sub-graph, and (4) between the vertices of different application anatomy sub-graphs. An example for (1) would be network interface controller components connected via interaction edges to store network telemetry. An example for (3) and (4) would be the operating-system processes of an application (3) or across multiple applications (4) connected via interaction edges to store telemetry on interprocess communication. Beyond these rules, the abstract model does not define any concrete interaction edges. Analogous to structure edges, we are adding interaction edges to the model definition in equation (4.1) as two distinct mapping functions for the platform and for application interaction[2]:

$$
\begin{aligned}
& s = (\ldots, s_{InteractAA}, s_{InteractPA}) \text{ where} \qquad (4.4)\\
& \quad s_{InteractAA} : E_{AA} \rightarrow V_{AA} \text{ assigning each application interaction edge } (●) \text{ its source } V_{AA},\\
& \quad s_{InteractPA} : E_{PA} \rightarrow V_{PA} \text{ assigning each platform interaction edge } (●) \text{ its source } V_{PA}.\\
& t = (\ldots, t_{InteractAA}, t_{InteractPA}) \text{ where}\\
& \quad t_{InteractAA} : E_{AA} \rightarrow V_{AA} \text{ assigning each application interaction edge } (●) \text{ its target } V_{AA},\\
& \quad t_{InteractPA} : E_{PA} \rightarrow V_{PA} \text{ assigning each platform interaction edge } (●) \text{ its target } V_{PA}.
\end{aligned}
$$

With interaction edges defined, we now have a complete definition for application and platform sub-graphs that can capture both, the structure and the interaction between application and platform vertices. However, the model does not yet allow us to define connections *between* application and platform sub-graphs. To solve this, we next introduce another type of edge to the model: allocation edges.

[2]The use of ... in equation (4.4) denotes that this is an extension of previous definitions

4.2.2 Allocations

Allocation edges allow us to model interaction between application and platform sub-graph vertices. Conceptually, a connection between the two means that an application is “running on the platform”, i.e., *consuming* platform resources, for example, an application process (E_{AA}) is consuming CPU and memory resources (E_{AA}). To reflect that allocation edges are not part of either the PA or AA sub-graphs, we add a new set of edges (E_{Alloc}) and corresponding mapping functions s_{Alloc} and t_{Alloc} to the definition in equation (4.1)[3]:

$$
\begin{aligned}
&E = (..., E_{Allock}) \text{ where} && (4.5)\\
&\quad E_{Allock} \text{ is a set of allocation edges, and} \\
&s = (..., s_{Alloc}) \text{ where} \\
&\quad s_{Alloc} \colon E_{Alloc} \rightarrow V_{AA\,j} \text{ assigning each allocation edge (●) its source } V_{AA}. \\
&t = (..., t_{Alloc}) \text{ where} \\
&\quad t_{Alloc} \colon E_{Alloc} \rightarrow V_{PA\,i} \text{ assigning each allocation edge (●) its target } V_{PA}.
\end{aligned}
$$

With the telemetry graph model extended to allocations, we can now represent the full telemetry graph as shown in figure 4.1: we can model the structure of applications and platforms, the interaction between components within applications and platforms, and, via allocations, the interaction between the two. Next, we will describe how time variance can be added to the model, i.e., how to capture changes in structure, interaction, and allocation over time, and how telemetry data is embedded in our graph model.

4.2.3 Time-Variance

Both application and platform structures are not fixed but change continuously throughout the evolution of the platform and applications. Consequently, the telemetry graph is time-variant, i.e., the relationship between edges as well as the existence of edges evolves with time. Platform providers might commission, decommission, or upgrade

[3] The use of ... in equation (4.5) denotes that this is an extension of previous definitions

compute nodes, add data storage, networking, or special-purpose compute (GPUs, FPGAs) capabilities of an HPC system during scheduled or unscheduled maintenance cycles. Advanced platform management systems might have the capability to change topologies dynamically during normal operations. Hardware and software failures might render a subset of nodes or a storage subsystem temporarily unavailable. All these are changes in structure that are of vital importance to applications that want to implement optimization and resilience strategies and hence must be captured in a telemetry information model. Similarly, application structure can change across multiple design iterations or different runs (e.g., with different numbers of processes), but also within the trajectory of a single execution[4]. Especially in non-monolithic, dynamic applications, changes in structure are quite commonplace and are a vital piece of information to interpret telemetry data. In order to capture time-variance in the telemetry graph, we add time index to the telemetry graph definition that describes the graph at a given time τ:

$$\begin{aligned} TG(\tau) = (V(\tau), E(\tau), s(\tau), t(\tau), \Sigma_V(\tau), \Sigma_E(\tau), \ell_V(\tau), \ell_E(\tau)) \text{ where} \\ \tau \text{ is a given instant in time} \end{aligned} \tag{4.6}$$

The temporally ordered graph sequence $TG = \langle TG(1)...TG(j)...TG(m)\rangle$ describes the evolution of the telemetry graph over time. A specific $TG(\tau)$ describes the platform and application components, their structure, and interaction at time τ.

4.2.4 Embedding Telemetry Data

With the abstract model definition in place, providing a well-defined structural and semantic context, we can now define how telemetry data is embedded into this model. We define three types of telemetry data: *properties*, *measurements*, and *events*. They are attached as *labels* on vertices (●,●), interaction edges (●,●) and allocation edges (●) (figure 4.2). Structure edges do not have telemetry data associated with them, as they represent the logical and physical structure of applications and platform at any given instant in time. Properties are a single value describing a vertex or an edge in

[4]For examples of applications with dynamically changing structures, see e.g., [Liang et al., 2020] and [Balasubramanian et al., 2016].

the graph. *node name* for example could be a property capturing the hostname of a node. Properties are mutable, meaning that they can change over time, but the change frequency is rather low. Measurements on the other hand are a continuous series of data points describing the behaviour of a vertex or an edge over time. A concrete measurement could for example be *Free Memory*, capturing the free memory available on a node over time. Measurements are immutable and can not be changed — only new data points can be added. The third type of data is event data. They describe an event associated with a vertex or an edge. For example, a *Terminated* event emitted from an application process. A single, mutable value is associated with each event.

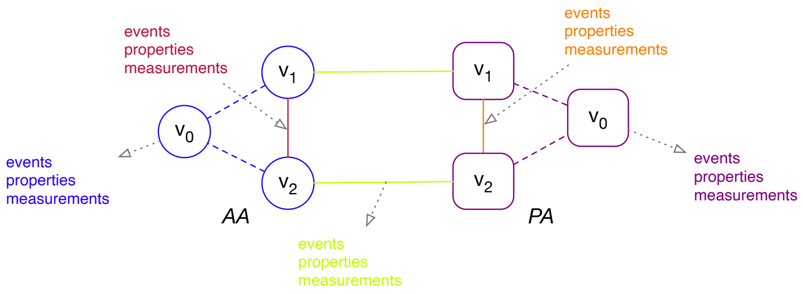


Figure 4.2: We define three types of telemetry data: *properties*, *measurements*, and *events* which can be attached as *labels* on vertices, and interaction and allocation edges.

For each type, we further distinguish between model-defined and user-defined data. Model-defined data is homogeneous across all vertices and edges of the same type. For example, if in a concrete model we define a `compute node` vertex type, we could define `node_name` and `num_cpus` as platform defined properties. It is then the responsibility of the telemetry platform to extract the values for these properties from the HPC system and update it across all *compute node* vertices. User-defined properties on the other hand do not have to be homogeneous across all vertices and edges of the same implementation type, only across those of a specific application anatomy graph under control by the user. For example, a user might define a custom property called *algorithm* on a *process* vertex that describes which internal algorithm variant was chosen by that specific process, or a *mesh density* property on a *thread* vertex representing the internal state of an HPC application.

Properties

Properties represent information that stays constant throughout the lifetime of a vertex or edge. Each element can have an arbitrary number of properties. We use graph labels to define properties within the telemetry graph

$$\begin{aligned}
&\Sigma_{VP}\ \mathit{and}\ \Sigma_{EP} \text{ finite set of available vertex and edge properties} \\
&\ell_{VP} : V \rightarrow \Sigma_{VP} \text{ map describing the properties of the vertices} \\
&\ell_{EP} : V \rightarrow \Sigma_{EP} \text{ map describing the properties of the edges}
\end{aligned} \tag{4.7}$$

This extends the definition of the telemetry graph to:

$$TG = (\Sigma_{VP}, \Sigma_{EP}, V, E, s, t, \ell_{VP}, \ell_{EP}) \tag{4.8}$$

A property (P) is defined by a *property key* (λ) and a *property value* (ρ), where property value is a single scalar type:

$$P = (\lambda, \rho) \tag{4.9}$$

Every concrete implementation of the telemetry graph defines a fixed set of property types for each vertex and edge type as the system-defined properties. For example, for a `RAM` vertex type, we could define `total` as a system-defined property type. These properties are guaranteed to exist for every vertex or edge of the same type and throughout the lifetime of the graph. They are continuously populated by the telemetry platform. An arbitrary number of user-defined property types can be defined ad hoc. User-defined properties are not guaranteed to exist consistently across vertices and nodes and throughout the lifetime of the graph. It is the user's responsibility to populate them with data.

Measurements

Measurements represent dynamic information that can be associated with any element of the telemetry graph. Each element can have an arbitrary number of measurements. Just like properties, measurements are defined as graph labels:

$$\begin{aligned}
&\Sigma_{VM} \textit{ and } \Sigma_{EM} \text{ finite set of available vertex and edge measurements} \qquad (4.10)\\
&\ell_{VM} : V \rightarrow \Sigma_{VM} \text{ map describing the measurements of the vertices}\\
&\ell_{EM} : V \rightarrow \Sigma_{EM} \text{ map describing the measurements of the edges}
\end{aligned}$$

This extends and completes the definition of the telemetry graph to:

$$TG = (\Sigma_{VM}, \Sigma_{VP}, \Sigma_{EM}, \Sigma_{EP}, V, E, s, t, \ell_{VM}, \ell_{EM}, \ell_{VP}, \ell_{EP}) \qquad (4.11)$$

A measurement (M) is defined by a *measurement key* (κ) and a *measurement value* (σ), where measurement value is a time-series:

$$\begin{aligned}
&M = (\kappa, \sigma) \qquad (4.12)\\
&\sigma = \{\sigma_t : t \in T\} \text{ where T is the (time) index set.}
\end{aligned}$$

Every concrete implementation of the telemetry graph defines a fixed set of measurement types for each vertex and edge type as the system-defined measurements. For example, for a `RAM` vertex type, we could define `available`, `used`, and `free` as system-defined measurement types. These measurements are guaranteed to exist for every vertex or edge of the same type and throughout the lifetime of the graph. They are continuously populated by the telemetry platform. An arbitrary number of user-defined measurement types can be defined ad hoc. User-defined measurements are not guaranteed to exist consistently across vertices and nodes and throughout the lifetime of the graph. It is the user's responsibility to populate them with the data.

Events

We distinguish between three different types of events: *structure events*, *interaction events*, and *allocation events*. Structure events capture the creation and destruction of platform and application entities, such as a new node being added to an existing cluster or a new process spawned by an application. Interaction events capture the

beginning and end of an interaction between two entities, and allocation events the beginning and end of an allocation of application to platform entities. Events consist of a *timestamp*, an *entity type*, and *event type*. The *timestamp* defines the time at which the event occurred. *Entity type* defines the (concrete) node or vertex type the event is associated with and *event type* is either *created* or *destroyed*. *Parent*, *from*, and *to* define the location of the new entity within the semantic structure.
Events are a key concept of the telemetry model as they capture the time-variant structure of the semantic graph. Through events, the semantic structure of platforms and applications is linked to and extracted from telemetry data. Creation events cause the insertion of new vertices and edges into the graph. Destruction events cause their removal.

4.3 Concrete Example

In this section, we demonstrate how the telemetry data model can be applied to the anatomy of a multi-tenant Beowulf [Becker et al., 1995] style HPC cluster architecture that builds on compute nodes of commodity computing and networking hardware (figure 4.3). We chose this type of architecture as it is probably the single most common system architecture found in the HPC landscape. Also, common to this architecture and included in this example is a shared filesystem that is available across all compute nodes and served by a central file server. On the system software architecture side, we assume a UNIX-style operating-system [Ritchie and Thompson, 1978] implementing a standard UNIX process model and batch workload manager managing access to the resources. To add some of the heterogeneity that is increasingly found in HPC cluster designs, we assume that a subset of compute nodes have GPU accelerators installed in order to support fast vector operations.

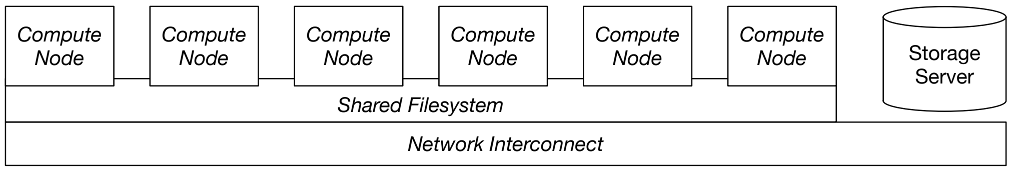


Figure 4.3: Example of a common *Beowulf*-style HPC cluster architecture with a shared filesystem and heterogeneous node configuration.

In order to get from the abstract model definition to a concrete implementation of a telemetry model description, we need to build a semantic model of the HPC cluster and application components, their structure and interactions. We then define the different vertex and edge types for the implementation model, i.e. the *structure edges* and *interaction edges* between the different vertex types within their respective graphs and the *allocation edges* between them. For each of these elements, we also define some sensible default properties and measures which will later be populated with telemetry data.

4.3.1 Platform Anatomy

The platform anatomy graph aims to capture the HPC cluster components and their structural relationships with each other. The telemetry information model does not set any constraints in terms of what the components should be and how they are connected. In some cases it might be sufficient to capture a platform architecture at the node level, in some other cases, capturing details down to the intra-node bus-level architecture might be required. If designed carefully, platform anatomy can be extended over time and additional levels of detail can be added without having to redesign the overall semantic structure of the graph.

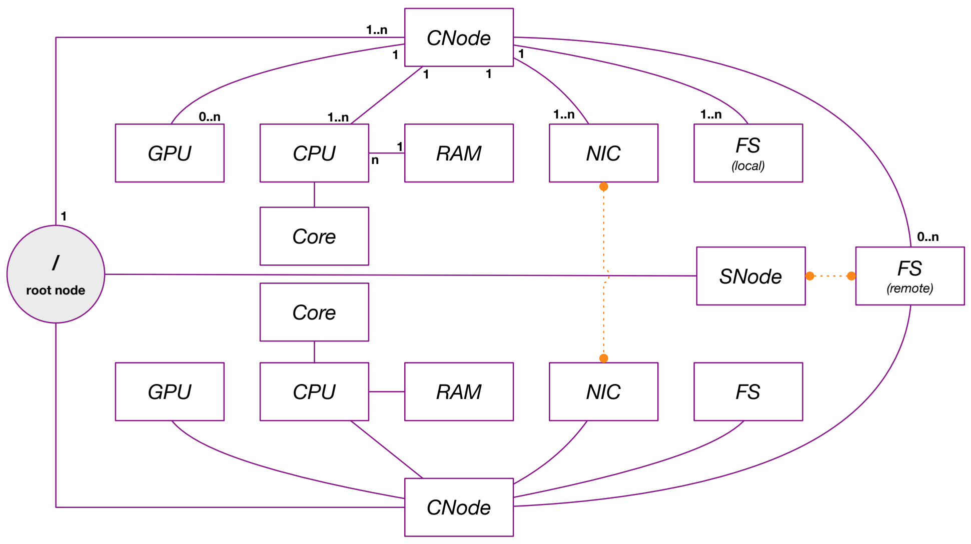


Figure 4.4: The vertex and edge types and their relationships defined for the platform anatomy graph. Multiplicities are denoted in ER notation.

The same holds true for the application anatomy. With the use cases for later evaluation in mind we define the platform anatomy vertex types as `ComputeNodes` (CNode), each consisting of a set of `CPU`s, `GPU`s and `Filesystems` (local and remote). Each `CPU` vertex is associated with a `RAM` vertex, in this implementation one `RAM` vertex for all `CPU` vertices on a node, reflecting a shared memory architecture. Furthermore, each `CPU` has one or more `Core` vertices. Network components are modelled via `NIC` (Network Interface Card) vertices which have *interaction edges* defined between them. These edges are used to capture network telemetry. Lastly, we define a `Filesystem` (FS) vertex type that models the UNIX mount points on a node. Filesystem edges can either be exclusive to one node (1-to-1 relationship) or shared between multiple nodes have an additional interaction edge with an `SNode` vertex which represents a storage server. `SNode` is a rather high-level abstraction and probably does not cover the structure and interaction required to model more advanced I/O and filesystem telemetry. It is however a good example of how a telemetry data model implementation can choose to reflect different levels of complexity in different areas. At a later time, one might decide to add additional vertex and edge types to model `FS`, `SNode` and their relationships in more detail.

4.3.2 Application Anatomy

HPC applications on any system architecture typically span a wide area of categories, ranging from tightly-coupled parallel applications to distributed workflows and service-oriented architectures. Each class of application has its own internal logical representation, concepts and building blocks. The application anatomy does not capture these explicitly. However, when these applications execute on a system, they all have the same physical building blocks and representation. For a UNIX-based distributed architecture like the one we are considering in this example, these are time-variant networks of communicating processes. Hence, we define `Process` as the main executable component. We furthermore define `CPUThread` as components linked together by a process and `Job` as the component linking together processes. To capture GPU processing, we also define a `GPUProcess` vertex. For the sake of simplicity, we ignore GPU threads in this example. Communication is modelled via `Socket` vertices. They correspond to the UNIX sockets that were opened and closed over time by each thread. The same concept is applied for `FileHandles` opened and closed by individual processes. Communication between threads is modelled as edges between `Socket` vertices. This gives us a simple application anatomy with five component types in a time-variant structure.

The data model describes application anatomy as a structure that exists independent of the application's execution state. In the case of this example, this means a graph of jobs, processes and threads that changes over time. Properties describing aspects of

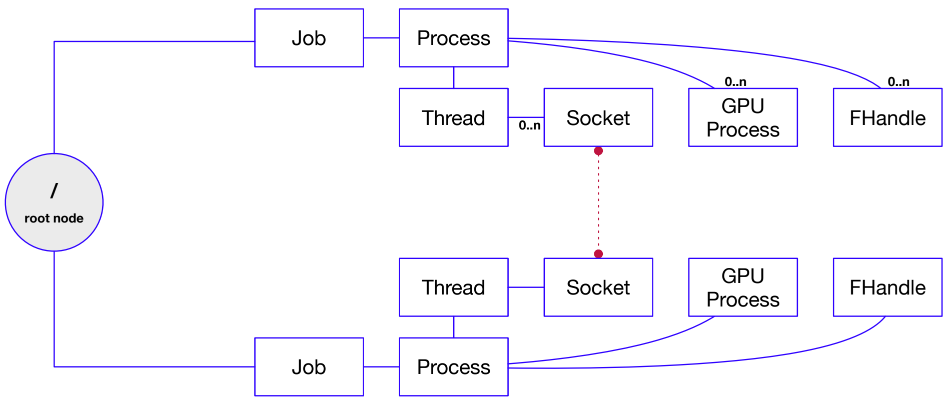


Figure 4.5: The vertex and edge types and their relationships defined for the application anatomy graph. Multiplicities are denoted in ER notation.

the execution of the application, i.e., the actual telemetry data such as memory or CPU cycle consumption are not captured as part of the application anatomy graph but as part of the mapping between application and platform anatomy. Default properties defined on the application anatomy level are therefore very limited. However, the application anatomy graph provides the structure in which all custom, user-defined application telemetry is embedded.

4.3.3 Allocations

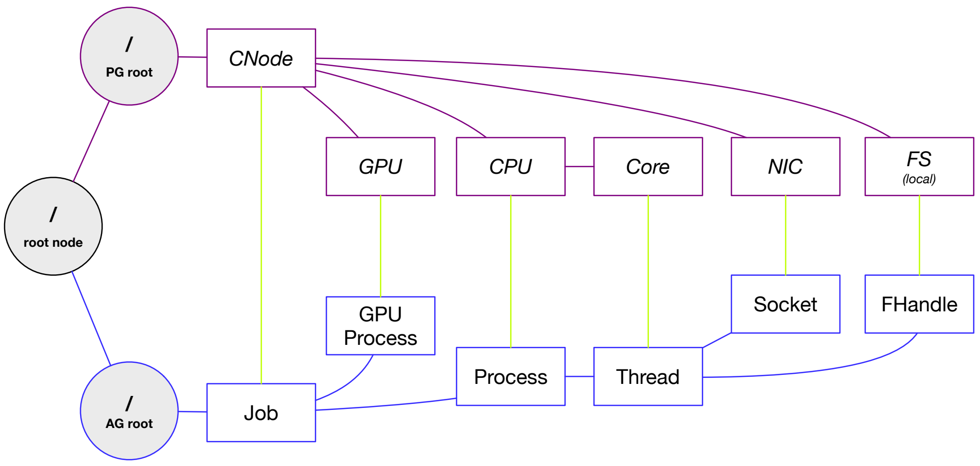


Figure 4.6: The allocation edge types connecting platform and application anatomy graph vertices.

Once an application has been launched and the jobs, processes, and threads come into "physical" existence, we can identify the allocation edge types between application and platform graphs (figure 4.6). Again, the allocation edge mapping follows the UNIX process model. We start with defining edge types between *Process* and *CPU* and between *Thread* and *Core*. Analogously, we define an edge type between *GPUProcess* and *GPU*. Allocations of communication channels are modelled as edges between *NIC* (Network Interface Card) and *Socket*. File allocations are modelled as edges between *FS* (Filesystem) and *FHandle* (File Handle).

4.4 Database Design

Graphs are common data structures that can be implemented on top of many storage systems, e.g., in-memory, object stores or a Relational Database Management System (RDBMS). We choose a Graph Database (GDB) as the implementation platform as it makes the translation from theoretical model to implementation very intuitive. Graph databases are optimized for traversing large graphs in constant time and hence contribute to the overall scalability and performance of the model implementation. While highly optimized for working with graph structures, graph databases are not well suited for storing and querying the large volumes of event and time seriesdata that are generated by system and application telemetry frameworks and systems. Time seriesdata has specific characteristics such as typically arriving in time order form, data is append-only, and queries are always over a time interval. While relational databases can store this data, they are inefficient at processing it as they lack important optimizations such as storing and retrieving data by time intervals. Consequently, we have chosen to use a dedicated Time-Series Database (TSDB) as a separate storage backend for measurements, i.e., time seriestelemetry, and cross-reference its content with the telemetry graph managed by the GDB. In this, we can realize efficient graph queries and at-scale time seriesanalysis within the same system.

Figure 4.7 shows the heterogeneous database architecture consisting of a GDB and a TSDB cross-referencing their data. Both storage systems have their own, domain-specific query language and interface. A *Semantic Ingest Service* consumes the steam of telemetry recorded on an HPC system and splits it into events and properties to be translated into graph structure and properties, and measurements to be translated into time series records. The added complexity of this heterogeneous setup can be hidden behind a *Consolidated Telemetry API*, which is described in more detail in chapter 5.

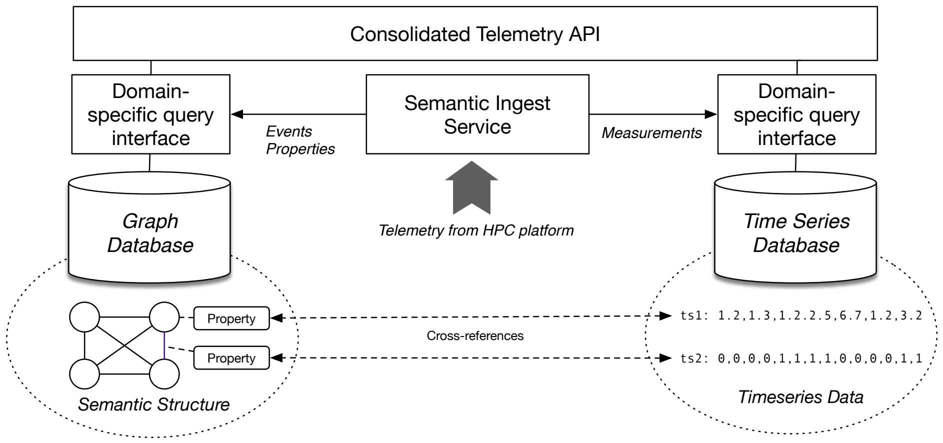

Consolidated Telemetry API
Domain-specific query interface
Semantic Ingest Service
Domain-specific query interface
Events
Properties
Measurements
Graph Database
Telemetry from HPC platform
Time Series Database
Property
Property
Cross-references
ts1: 1.2,1.3,1.2.2.5,6.7,1.2,3.2
ts2: 0,0,0,0,1,1,1,1,0,0,0,0,1,1
Semantic Structure
Timeseries Data

Figure 4.7: The database design consists of two separate database systems for semantic graph data and time seriesdata.

4.4.1 Graph Database

We use the Neo4j [Webber, 2012] graph database for the evaluation of the database design. Neo4j is an ACID-compliant transactional database with native graph storage and processing capabilities that support the labelled property graph and W3C's RDF graph models, and their respective query languages, the *Apache TinkerPop Gremlin* and *Cypher* graph traversal language and the W3C standard *Resource Description Framework's* (RDF) *SPARQL* query language. We chose Neo4j for its native support for the labelled property graph model as this provides an intuitive mapping for our telemetry model. In principle, any other graph database can be used to store the telemetry graph. [Fernandes and Bernardino, 2018] provides a good overview and comparison of graph database systems.

In Neo4j, everything is stored in the form of an edge, node, or attribute. Each node and edge can have any number of attributes. Both nodes and edges can be labelled. Labels are used to narrow searches in graph queries. In graph-theoretical terms, a labelled property graph is defined as a directed, vertex-labelled, edge-labelled multigraph with self-edges, where edges have their own identity. Property graphs generally use the term node to denote a vertex, and relationship to denote an edge. A property graph defines the following elements:

1. **Entity:** There are two types of entities: *Nodes* and *Relationships*. An entity has a unique, comparable identity that defines whether two entities are equal. It is assigned a set of *Properties*, each of which are uniquely identified in the set by their respective *Property Keys*.

   (a) **Node:** A node is the basic entity of the graph, with the unique attribute of being able to exist in and of itself. It may be assigned a set of unique *Labels* and have zero or more incoming and outgoing *Relationships*.

   (b) **Relationship:** A relationship is an entity that encodes a directed connection between exactly two nodes, the source *Node* and the target *Node*. An outgoing relationship is a directed relationship from the point of view of its source node. An incoming relationship is a directed relationship from the point of view of its target node. A relationship is assigned exactly one *Relationship Type*.

2. **Token:** A token is a non-empty string of Unicode characters. There are three types of tokens: *Labels*, *Relationship Types*, and *Property Keys*.

   (a) **Label:** A label is a token that is assigned to *Nodes* only.

   (b) **Relationship Type:** A relationship type is a token that is assigned to *Relationships* only.

   (c) **Property Key:** A property key is a token which uniquely identifies an *Entity's* property.

3. **Property:** A property is a pair consisting of a *Property Key* and a *Property Value*. A property value is a concrete, scalar type or a list of concrete, scalar types.

The mapping from the elements in our abstract model definition to the elements of the property graph is trivial. We define the root vertex of the semantic graph as a *Node* with the *Label* `Root`. Vertices of the platform and application anatomy graphs are defined as *Nodes* with the *Labels* `PAGNode` and `AAGNode`. Structure edges between both types are defined as *Relationship* with the *Relationship Type* `STRUCTURE`. Interaction edges are defined as *Relationship* with the *Relationship Type* `INTERACTION`. Allocation edges are defined as *Relationship* with the *Relationship Type* `ALLOCATION`. The time-variance of all nodes and relationships is modelled via two *Properties* with the *Property Keys* `created` and `destroyed`. We call this definition the domain model of the graph database.

[5] `https://neo4j.com/developer/graph-visualization/`

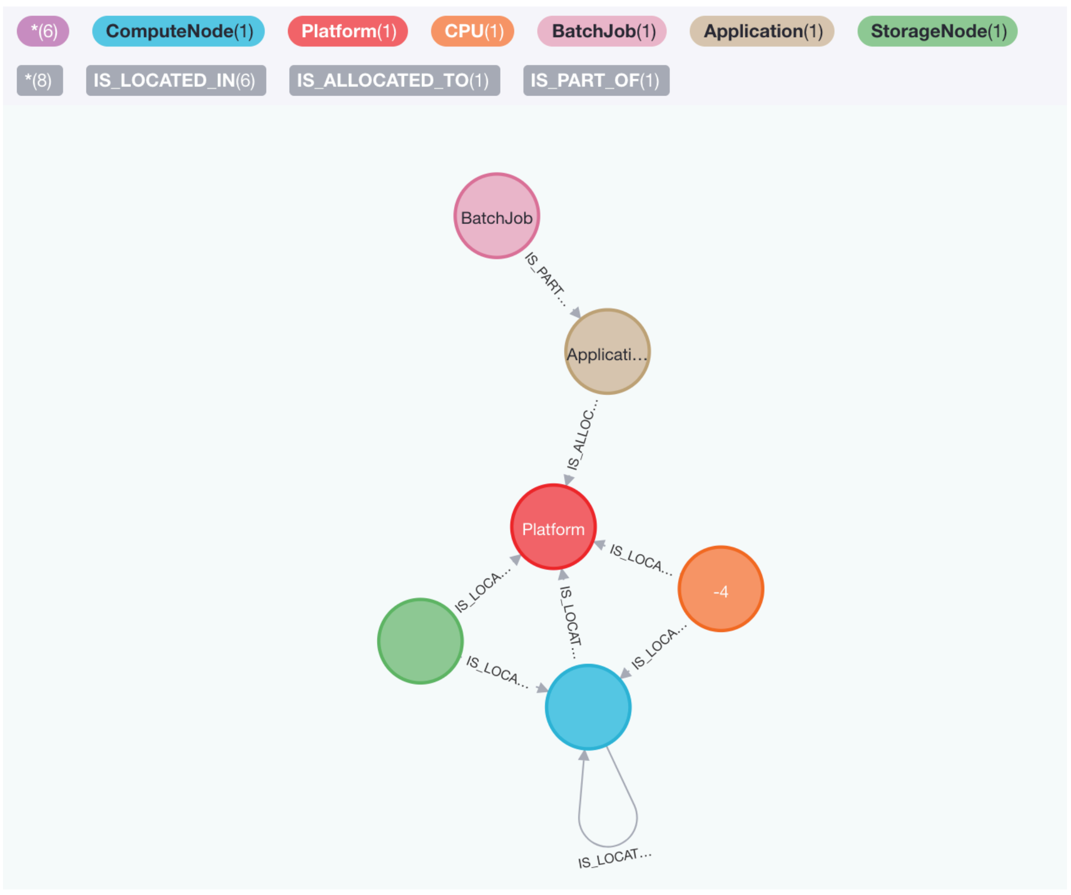


Figure 4.8: The database domain model for the concrete model realization example from the previous section. This graph was generated automatically by the interactive Neo4j query editor[5].

Since the property graph model does not differentiate between abstract and concrete model, we fold the definition of the concrete model realization (in this case from the model defined in the previous section) into the definition of the abstract model above. The different AAG and PAG node types are added as additional *Labels* to `PAGNode` and `AAGNode` *Node*, e.g., `CNODE`, `CNODE`, `CPU`, and `PROCESS`. Structure, interaction and allocation edges are not further qualified beyond the relationship types `STRUCTURE`, `INTERACTION`, and `ALLOCATION`. Further, qualifying the specific types of structure, interaction, and allocation relationships remains a future option. Since each relationship can only be of one relationship type, further qualification would have to be modelled via a prefix / suffix scheme, e.g., `INTERACTION_SEND_RECIEVE`. The complete domain model is shown in figure 4.8.

With the graph database domain model in place, we can now generate a dynamic property graph from telemetry and events collected on an HPC system. The details how the graph is populated, i.e., how the data collected on a system is translated into nodes, relations, and properties is described in section section 5.2.1. Using a graph query language, it is now possible to explore the structure of the HPC system and its applications.

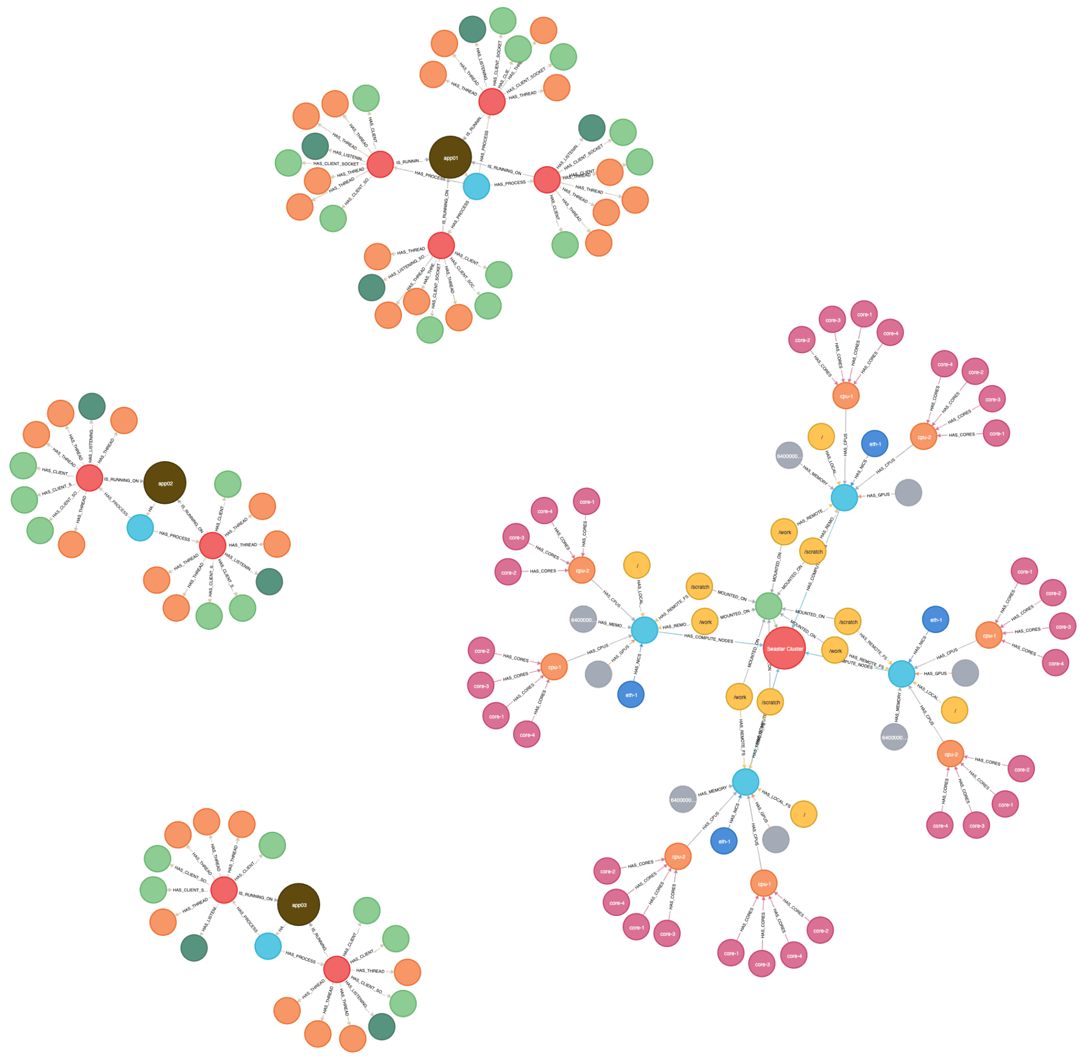


Figure 4.9: Sample telemetry graph (truncated) consisting of one platform anatomy (bottom right) and three application anatomy sub-graphs. Only vertices and structure edges are shown. This graph was generated automatically by the interactive Neo4j query editor.

Querying the Graph

For property graph traversal, Neo4j provides support for two declarative query languages: *Cypher* and *Gremlin*. Gremlin is more powerful as it gives more fine-grained control over defining the exact traversal pattern for a query, whereas in Cypher the engine tries to find the best traversing solution itself. For the illustrative use cases in this chapter, we use *Cypher* due to its simplicity. In Cypher, the entire telemetry graph can be traversed with a simple query across all vertices and edges as illustrated in listing 11. This query returns all platform- and application vertices, structure-, interaction-, and allocation edges that are defined in the telemetry graph. The graph shown in figure 4.9 visualizes this query using a sample data set consisting of a four-node cluster and three separate applications.

```
MATCH (n) OPTIONAL MATCH (n)-[r]-() RETURN n,r
```

Listing 11: Cypher query for a full telemetry graph traversal returning all platform- and application vertices, structure-, interaction-, and allocation edges. The visual result is shown in figure 4.9.

Similar to SQL, Cypher queries can be further specified using a `WHERE` clause. The example in listing 12 shows a Cypher query that returns a subset of the telemetry graph consisting of all operating-system processes that are currently allocated to the compute node labelled `node01`.

```
MATCH  (node1:CNode {hostname: "node01"})-[:ALLOCATED_TO]-(proc:Process)
WHERE  proc.destroyed = 0
RETURN proc
```

```
{"pid":"pid-62563"}
{"pid":"pid-62543"}
{"pid":"pid-62537"}
{"pid":"pid-62532"}
```

Listing 12: Cypher query returning all active (non-destroyed) processes that are currently allocated to the compute node labelled `node01`. The lower part of the listing shows the result set, i.e., the four process IDs that match the query.

4.4.2 Time-Series Database

The telemetry graph domain model defined in the GDB stores the semantic structure itself and the properties of its entities. Since measurements are in essence time serieswe use a dedicated TSDB for storing them. The added complexity of splitting the labels of the telemetry graph across two different systems is offset by much more efficient querying and analysing of time-series data, a critical capability of any telemetry platform. As a database for time seriesdata, we have chosen InfluxDB [6], an open-source, highly-available storage and retrieval platform for time seriesdata. InfluxDB is used in fields such as operations monitoring, application metrics, Internet of things (IoT) sensor data, and real-time analytics. We chose InfluxDB primarily due to our previous experience with the system and for its surrounding ecosystem of tools, such as the Telegraf server agent which we use for telemetry data ingestion in our prototype (see section 5.2.1). Any other database system that is optimized for handling time seriesdata can in principle be used instead. Especially SaaS-based databases, such as AWS Timestream[7] for example, could provide an interesting alternative for cloud-based deployments. [Bader et al., 2017] provides a good overview of time seriesdatabase systems.

InfluxDB's data structure is based on the concept of *points*. A point has four components: a *measurement*, a *tagset*, a *fieldset*, and a *timestamp*: The measurement provides a way to associate related points that might have different tagsets or fieldsets. The tagset is a dictionary of key-value pairs to store metadata with a point. The fieldset is a set of typed scalar values — the data being recorded by the point. In the example in listing 13, the measurement is `cpu_frequency`. The tagset is `hostname=node01` and `cpu_id=1`. The keys, `hostname` and `cpu_id`, in the tagset are called tag keys. The values, `node01.seastar` and `1`, in the tagset are called tag values. The fieldset is `max=3500.00,min=1600.00,current=3500`. The keys, `max`, `min`, and `current` in the fieldset are called field keys. The values, `3500.00`, `1600.00`, and `3500.00`, in the fieldset are called field values.

InfluxDB provides the SQL-like InfluxQL query language [8] to query time seriesdata. Listing 14 shows a simple conditional query of the `cpu_frequency` measurement. The result of the query is show in listing 15.

```
cpu_freqency,hostname=node01,cpu_id=1 max=3500.00,min=1600.00,current=2600
cpu_freqency,hostname=node01,cpu_id=1 max=3500.00,min=1600.00,current=2600
cpu_freqency,hostname=node01,cpu_id=1 max=3500.00,min=1600.00,current=3500
```

Listing 13: Example of a series of InfluxDB CPU frequency measurement with two tags (hostname, cpu_id) and three field keys (min, max, current). Timestamps are omitted.

```
SELECT current FROM cpu_frequency
  WHERE "hostname" = "node01"
  LIMIT 3
```

Listing 14: Example InfluxQL query selecting the `current` key from the fieldset of the `cpu_frequency` measurement.

```
name: cpu_frequency
--------------
time                    current
2020-08-18T00:00:00Z    2600
2020-08-18T00:00:05Z    2600
2020-08-18T00:00:10Z    3500
```

Listing 15: The result of the query in listing 14 lists the current CPU frequencies recorded at `time`.

Beyond these simple examples, InfluxQL provides a rich set of functionalities that are specifically designed to support working with time series. This includes *mathematical functions* and *continuous queries*. Mathematical functions provide a rich set of query functions for aggregating, selecting, transforming, and predicting data. Examples are the `SAMPLE()` function which returns a random sample of N field values using reservoir sampling, and the `HOLT_WINTERS()` function which returns N number of predicted field values using the Holt-Winters [Chatfield, 1978] seasonal method. Continuous queries are queries that run periodically in the background on real-time data and store query results in a specified measurement. They can for example be used to shorten query runtimes by pre-calculating expensive queries or to automatically downsample commonly-queried, high-precision data to a lower precision.

[6]https://www.influxdata.com/
[7]https://aws.amazon.com/timestream/
[8]https://github.com/influxdata/influxql

4.5 Summary

In this chapter, we have developed a telemetry data model that at its core supports the temporal variability of HPC system and application structures. These topological, temporal and spatial properties along with the also temporally variable mapping of application structure to platform structure sets this model apart from other, existing telemetry data models. In the next chapter, we incorporate the hybrid GDB / TSDB database design into the larger context and components of a telemetry platform prototype for further evaluation.

Chapter 5

Implementation and Evaluation

In this chapter, we discuss SEASTAR, a prototype implementation of the telemetry platform concept and its integration with an HPC cluster. The first part of this chapter describes the telemetry platform architecture, which is built around the heterogeneous telemetry graph implementation introduced in the previous chapter. Development and evaluation of SEASTAR takes place in a cloud-based environment in Amazon Web Services which we use both for SEASTAR and the HPC cluster testbed that we have built to explore the integration between telemetry platforms and HPC clusters. The second part of this chapter focuses on the details of this development environment and integration. In the last part of this chapter, we demonstrate SEASTAR's capabilities by implementing a machine learning-based application anomaly detection service.

5.1 Requirements and Constraints

The functional requirements introduced in section 3.1.3 set very few constraints for how a telemetry platform would be implemented in practice. It specifically states in *R.F14*, that "the telemetry platform, its interfaces, data model and capabilities must be described independently of its implementation". The previous chapter provides the high-level specification for the data model. The SEASTAR architecture in section 5.2 provides a conceptual description and an informal specification of the architecture of a telemetry platform and its components. In section 5.3, we then add additional context and constraints to narrow-down the implementation pathway for our specific implementation prototype. Firstly, we define a series of non-functional requirements like scalability to make sure that the implementation has relevance for a real-life HPC context with thousands of nodes and tens- or even hundreds of thousands of telemetry signals per second flowing through the system. Secondly, we define the non-functional requirement — or constraint — that the implementation effort has to be feasible in the context of this work, while still being relevant enough to evaluate some of our core concepts and research questions. While this is a difficult balance to maintain, modern cloud technology and an abundance of ready-to-use open-source software packages have the potential to simplify implementation complexity significantly and to keep the prototype implementation manageable.

5.1.1 Non-Functional Requirements

In contrast to functional requirements that we defined in section 3.1.3, the non-functional requirements introduced here specify criteria related to the operation of a telemetry platform, rather than its specific behaviour or functions.

R.NF1 (Dynamically) Scalable

To support current and future large-scale, potentially peta- and exascale HPC systems and applications, a telemetry platform must be scalable. This means that all platform components must be designed from the ground up in a way that they can cope with millions of telemetry data points per second. Ideally, a telemetry platform can scale up and down *dynamically*, i.e., retain and release infrastructure resources based on its current load in order to minimize its overall cost of operation. Scalability, i.e., the ability to "keep up" with the scale of current and future HPC systems is the single most important non-functional requirement determining the success of a telemetry platform.

R.NF2 Non-Invasive

To support existing HPC systems and applications, the telemetry platform must be non-invasive, i.e., it must not require mandatory changes to platforms and applications. It must also be possible to integrate the telemetry platform with existing system software, including workload managers and batch schedulers.

R.NF3 (User-Defined) Responsiveness

To support real-time use cases, a telemetry platform should provide data in a time-line manner, i.e., the delay between the generation of a data point, and the data point becoming available in the platform should be minimal. Since provisioning data in real-time is resource-intensive, the degree of responsiveness required would ideally be controllable by the applications based on their requirements.

5.1.2 Implementability in a Research Context

Building a complex distributed system that scales to the extent required for ingesting, processing, and storage of HPC telemetry from scratch can be a non-trivial effort. One important requirement for the telemetry platform implementation was to keep the effort manageable within the constraints of this work. At the same time, we still want to give a direction for the architecture and technology choices for a future large-scale production implementation of the telemetry platform concept. The two design principles derived from this requirement are:

1. **Use existing (open-source) software wherever possible.** Many of the capabilities needed by an HPC telemetry platform can be provided by existing software packages. This way, building a prototype becomes more of a composition and configuration exercise than a big software development effort.

2. **Use public cloud services.** Similar to using existing software components, using cloud services can significantly reduce implementation complexity of the prototype. While HPC clusters typically do not run on public cloud resources, a telemetry platform can still be operated outside the HPC cluster's data centre as long as latency and bandwidth are sufficient.

As previously discussed, an HPC telemetry platform is functionally very similar to the large-scale data analytics and processing platforms commonly found in industry IoT applications. A rich body of practitioner's knowledge on how to build these platforms is available online. Many of the cloud architecture and technology choices made in this chapter were directly influenced by those. In addition, a plethora of commercial vendors for data platform capabilities exist. While these might be interesting for telemetry platform implementations in a more commercial context, they are out of scope for this prototype. Furthermore, the majority of commercial offerings are built around freely available open-source technologies.

5.2 The SEASTAR Architecture

SEASTAR is an architecture and prototype implementation of a telemetry platform that builds on the heterogeneous GDB/TSDB database architecture introduced in the previous chapter. It provides a context in which we can quickly validate concepts, rather than aiming to be a production-ready, stable reference implementation. Nonetheless, SEASTAR provides all relevant components to run end-to-end use case evaluations. SEASTAR follows a microservice implementation architecture approach, i.e., it provides self-contained functionality as independent services, exposed via APIs instead of a single, monolithic service. While microservice architectures demand a certain amount of development and engineering overhead, the benefits of this approach become apparent when we want to explore scalability and composition. The modular architecture allows us to explore the scalability individually for each of the components with minimal interference from the overall system. From a composability perspective, it allows us to explore the effects of different configurations and granularities of components easily, e.g., running one telemetry service end-point per node v.s., one end-point for multiple nodes.

SEASTAR services are written in the Python programming language and expose their functionality via REST APIs. In order to minimize the complexity of the implementation, the design philosophy of SEASTAR is to reuse whatever capabilities are available as open-source software. Consequently, the majority of services are just thin wrappers around existing software. For example, we re-use the native data query and processing services of the InfluxDB and Neo4j databases wrapped into a SEASTAR programming framework that is aligned with SEASTAR's "look and feel" and unified authorization and access mechanisms.

SEASTAR consists of five logical components: ingestion, storage, processing and querying, workspaces, and application sidekicks (figure 5.1). Each of the components consists of one or more individual services. The telemetry ingestion component is responsible for inserting system and application telemetry correctly into the semantic context of the telemetry graph. The telemetry storage component manages the physical storage and access to the telemetry graph which is accessed via the processing and querying component. The application sidekick component provides capabilities for building and hosting telemetry-based application support services, and the workspace component provides capabilities for provisioning data science and visualization tools to interactively explore and work with telemetry data. The remainder of this section

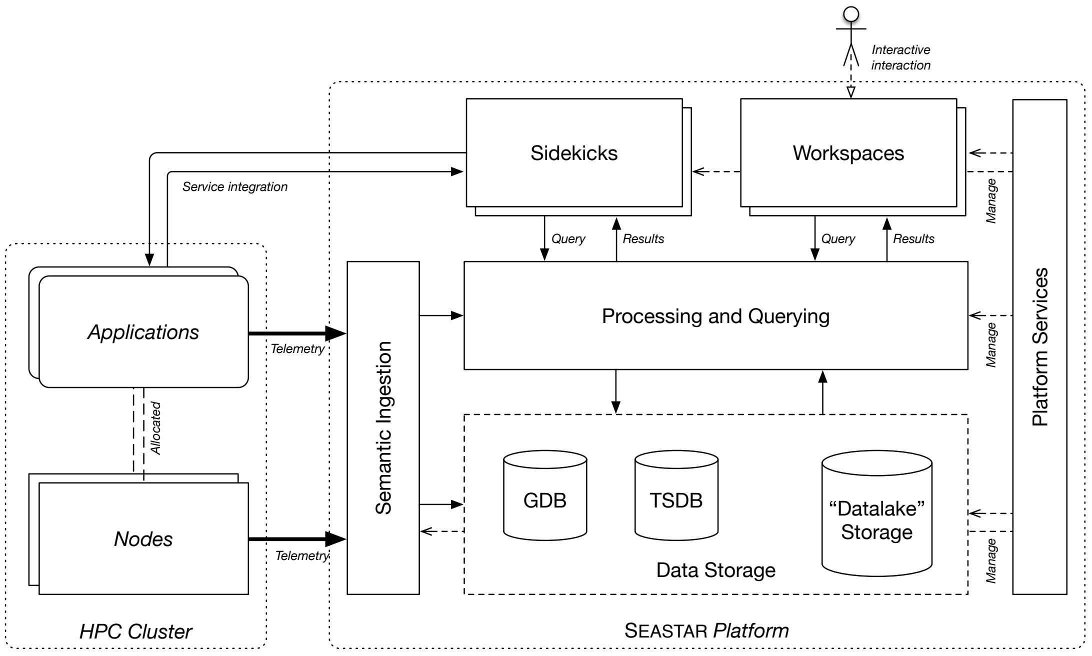


Figure 5.1: Logical architecture showing the functional components of the SEASTAR platform, ingestion, storage, processing and query, application sidekicks, and workspaces, orchestrated and managed by a set of platform core services. The two modes of interaction with the platform are *interactive* via workspaces and *integrated* via application sidekick services.

describes the six components that comprise the SEASTAR telemetry platform and the services and interfaces they provide.

5.2.1 Telemetry Ingestion

The ingestion component (figure 5.2) is responsible for moving telemetry data from the HPC cluster's compute nodes into the semantic structure of the telemetry graph. The basic principle is that of an Extract, Transform, Load (ETL) process, i.e., the procedure of copying data from one or more sources into a destination system which represents the data differently from the sources or in a different context than the sources. Here, we extract telemetry data, transform it into graph operations and time seriesdata, and load it into the graph and time seriesdatabases.

Three sub-components are comprising telemetry ingestion: *Collection Agents* that extract telemetry from platform and applications, the *Transport and Caching Layer*, and the *Semantic Ingestion* service which inserts the telemetry data into the logical telemetry graph split across graph and time seriesdatabases. Next, we describe the

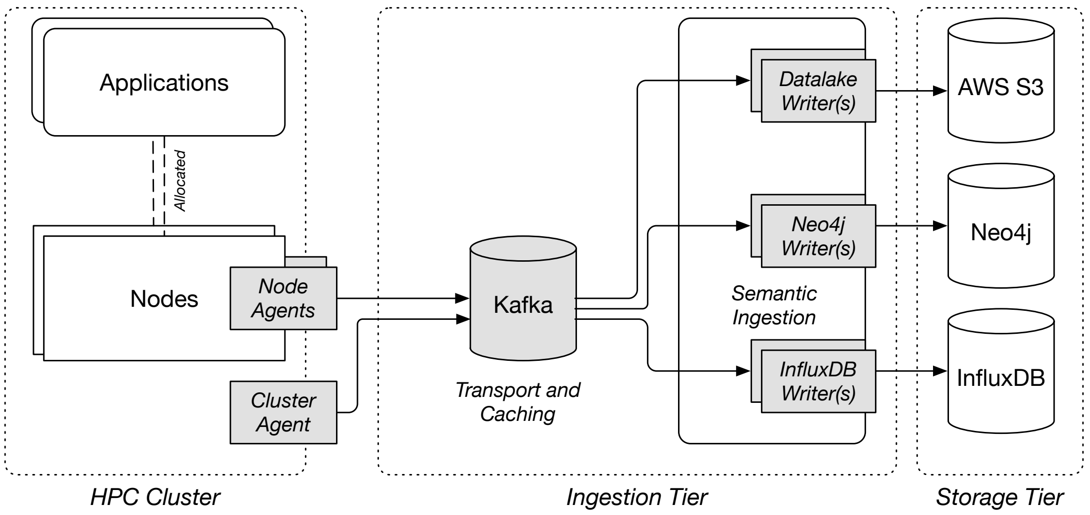


Figure 5.2: The ingestion component (grey) collects telemetry data via *Collection Agents* that are installed on HPC cluster nodes and writes it via a distributed event-streaming service (Kafka) to the graph (Neo4j) and time series(InfluxDB) databases.

implementation details of the sub-components.

Collection Agents

Data is collected via two services, the *Node Agents* and the *Cluster Agent*. Node agents are installed on each compute node of an HPC cluster and collect system and application telemetry via standard operating-system interfaces (see section 2.1). The cluster agent runs only on one node of the cluster, typically the head- or login-node, and collects data that is not available through operating-system interfaces on compute nodes, such as queueing system state and job allocations.

For the node agent, we use *Telegraf* [InfluxData, 2020a], an open-source server agent that is capable of collecting operating-system and, via input plug-ins, application-specific telemetry. We have chosen Telegraf based on familiarity and due to its proliferation in large-scale cloud and distributed systems and infrastructure monitoring applications. Many alternatives exist. Especially the Lightweight Distributed Metric Service (LDMS) [Agelastos et al., 2014] that was developed at Sandia National Laboratories to continuously capture system and applications profiling data on supercomputers will be an interesting alternative for a production system deployment of SEASTAR. In addition to the standard Telegraf host-metrics plugin, we use the *Procstat* [InfluxData, 2020b] plug-in to monitor system resource usage of individual processes using their /proc data. *Procstat* transmits IO, memory, CPU, and file descriptor-related measurements. Via several output plug-ins, *Telegraf* can send the collected telemetry to different targets, such as files, databases and streaming APIs. Connectivity with the transport and caching layer is provided via the Kafka output plug-in. The plug-in uses the standard Kafka binary line protocol over a TCP connection. The *Telegraf* data structure looks as follows:

- **Timestamp:** Date and time associated with the fields.
- **Tags:** Key/Value string pairs used to identify the metric.
- **Metric name:** Description and namespace for the data.
- **Fields:** Key/Value pairs that are typed and contain the metric data.

While the *Node Agents* are responsible for collecting measurements about hardware and applications on compute nodes, the *Cluster Agent* is responsible for collecting the more overarching, structural information about an HPC cluster. The *Cluster Agent* is designed as a standalone service written in Python that is deployed on the head node of our testbed cluster (see section 5.3 for more details on deployment). The main source for structural information is the command-line interface to the SLURM workload management software that we use in our testbed cluster. Through the SLURM command-line, information about jobs, queues, and nodes is available. In many real-world HPC clusters, other, often vendor-specific cluster management tools can provide much more in-depth information, not just about compute nodes, but also about the state and configuration of other hardware, such as interconnect and network hardware. Future versions of the *Cluster Agent* can be extended to incorporate those. Alternatively, the *Cluster Agent* could be realized as a *Telegraf* input plug-in to simplify and homogenize the SEASTAR architecture further. The current implementation of the *Cluster Agent* utilizes the Kafka Python client library [Powers, 2020] to send its data to the transport and caching layer.

Semantic Decomposition

The key contribution of SEASTAR is to provide an implementation architecture that can capture both, time seriesdata, and the structural, semantic framework in which it is embedded. In order to achieve this, the data collected by the agents must be split up into time seriesdata, i.e., *measurements* and structural information, i.e., *events* at collection time before the information is passed on the transport and storage layers. In an early prototype of SEASTAR, we implemented this decomposition step centralized as part of the *Data Writer* components, however, the scalability of this approach was not satisfactory with a growing number of nodes. As a consequence, the current version of our prototype implements semantic decomposition within the Agents to achieve better scalability.

At collection time, the three different types of telemetry data defined in section 4.2.4 are identified in the telemetry data stream: events, properties, and measurements. Events are further categorized as allocation, structure, and interaction. Events are collected, both explicitly and implicitly. For example, a data package describing a process memory consumption measurement might contain a new process ID (PID) tag for the first time, which eventually translates to a "new process" structure event.

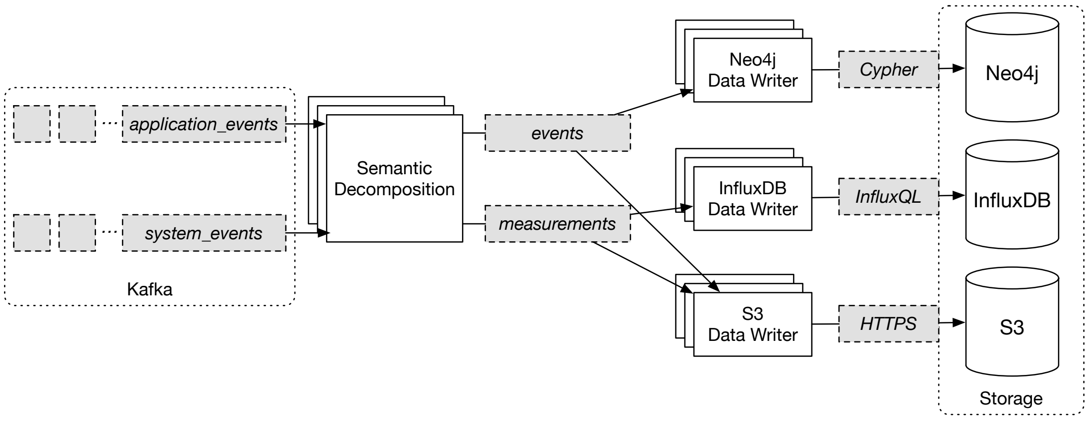


Figure 5.3: This diagram shows the data-flow and transformation through the semantic ingestion components: data streams arrive as Kafka topics, are translated into events and measurements by the *Semantic Decomposition* service, that are then inserted into Neo4j and InfluxDB via the *Data Writers*. A separate *S3 Writer* stores all events and measurements in the telemetry data lake.

Transport and Caching

Once the data has been collected and tagged, it is sent via a TCP binary protocol to the caching layer which consists of one or more Kafka services that relay the data further towards the storage layer. Kafka [Kreps et al., 2011] is a scalable distributed publish-subscribe event-streaming service. It implements the "message set" abstraction that naturally groups messages together to reduce network roundtrip overhead. This leads to larger network packets, larger sequential disk operations and contiguous memory blocks which allows Kafka to turn a bursty stream of random writes into linear writes. The rationale behind introducing Kafka as transport and caching layer instead of ingesting telemetry data directly to the storage layer is three-fold:

1. **Enable Scalability**: When building a telemetry platform for large-scale HPC systems, scalability of the overall implementation is the single most important concern from a non-functional requirement perspective. A key to enable scalability is to ensure that all components of the telemetry data platform are individually scalable. To support scalable components like databases, the data transport layer must be scalable as well. Kafka provides this scalability through clustering, i.e., by adding additional service instances.

2. **Fault-tolerance**: Failure and errors are intrinsic to distributed systems and the error rate statistically increases with the size and complexity of the system. Without a caching mechanism between data producers and consumers, a failure in the semantic ingestion component or further down at the storage tier would inevitably lead to data loss. Kafka can be configured with large disk storage backends that can buffer minutes or even hours of telemetry data and transparently release to its consumers once they become available again.

3. **Architectural flexibility**: The generic publish-subscribe data-transport mechanism Kafka provides allows the addition of new data sinks and sources without having to re-architect or reconfigure the overall system. This provides great flexibility for exploring and implementing alternative data-flow architectures. For example, future use cases might require data ingestion into an alternative storage system or allow HPC services to directly consume from, or contribute to the telemetry data stream.

Kafka organizes and durably stores events in so-called *topics*. A topic can be seen as folder in a filesystem, and the events are the files in that folder. Topics in Kafka are always multi-producer and multi-consumer: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to those events. Events are not deleted after consumption and a topic can be read as often as needed. For each topic, a retention time is set that defines for how long Kafka should retain the events. Kafka's performance is effectively constant [Kreps et al., 2011] with respect to data size, so storing data for a long time does not impact the performance. Topics are *partitioned*, meaning a topic can be spread over a number of different Kafka service instances. This distributed placement is important for scalability because it allows client applications to both read and write the data from/to many service instances at the same time. For this prototype implementation, we have configured two separate Kafka topics: `application_events`, which contain all application telemetry, and `system_events`, which contain all system telemetry. This allows us to split up the semantic ingestion logic for each stream.

5.2.2 Data Storage and Management

Once the telemetry data has been collected and processed by the semantic ingestion component, it flows into the storage layer of the telemetry platform. In the previous chapter, we have described how time seriesare stored in a separate TSDB and cross-referenced with the rest of the telemetry graph that is stored in a GDB. Both of these database systems are purpose-built and can cater to real-time or near real-time data access and query requirements. However, both databases are not designed to execute massive and complex queries across large amounts of historical data or bulk data access as it is required for applying machine learning techniques to telemetry data. Data storage in these database systems is also expensive, as the data is held either directly in memory or on fast hard disks in order to allow for short query times. In order to address the cost aspect and to support bulk-data access use cases, we introduce a secondary storage tier, we call the *Telemetry Data Lake* as shown in figure 5.4.

SEASTAR uses the Amazon Simple Storage Service (AWS S3), an inexpensive cloud-based object store as the storage technology for the *Telemetry Data Lake*. The storage units of Amazon S3 are objects which are organized into *buckets*. Each object is identified by a unique, user-assigned key. Buckets and objects can be managed using the AWS SDK or with the Amazon S3 REST API. Objects can be up to five terabytes in size with up to two kilobytes of metadata. There are no size restrictions on buckets. AWS S3 furthermore integrates directly with several large-scale data processing services in the AWS ecosystem, such as *Elastic Map Reduce* (EMR) and the *Athena* interactive query service, which makes it a good candidate as secondary storage for telemetry data.

This bi-modal storage and data processing architecture is an implementation of the *Lambda architecture* [Kiran et al., 2015], an architecture that is designed to handle massive quantities of data by taking advantage of both batch and real-time-processing methods. It aims to balance latency, throughput, and fault-tolerance by using batch

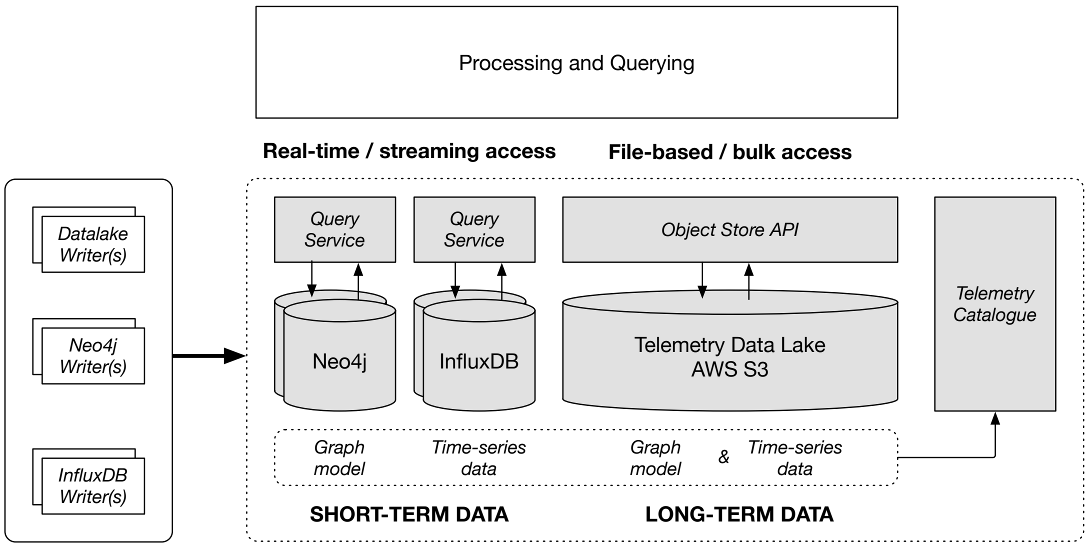


Figure 5.4: SEASTAR's bi-modal storage and data processing architecture (lambda architecture): telemetry is replicated into two different storage backends to cater for both, real-time streaming and bulk processing use cases.

processing (AWS S3) to provide comprehensive and accurate views of batch data, while simultaneously using real-time processing (Neo4j, InfluxDB) to provide views of online data. Lambda architectures depend on a data model with an append-only, immutable data source. It is intended for ingesting and processing timestamped events with a natural time-based ordering that are appended to existing events rather than overwriting them. A coherent state across batch- and real-time storage and processing systems is determined implicitly from the natural time-based ordering of the data. This makes the lambda architecture a good storage and data processing architecture for HPC telemetry data.

The storage space requirements depend on four factors: (1) the number of distinct telemetry measures collected, (2) the sampling frequency, (3) the size of the HPC platform, and (4) the data retention time. For a full set of around 400 operating-system and process metrics on a 128-node cluster, we calculate about 4 GB of raw data in 24 hours with a 10-second collection frequency. For a large cluster such as EPCC's ARCHER2[1], a 5,848-node HPE Cray EX supercomputer, this translates to 128 GB of raw data per day or 46 TB of raw data per year. By today's standards, these are rather modest storage requirements for a cloud-based object storage system, however, increasing the sampling frequency, e.g., to 1 second will increase the data volume by

[1]https://www.epcc.ed.ac.uk/archer20

and order of magnitude. The numbers we report here refer to raw, i.e., uncompressed data. Data volume can be effectively controlled and reduced by applying compression techniques and, equally important, by smart data management processes that control retention time, especially for user-generated telemetry and allow for dynamic and selective sampling frequency control. For example, a mechanism be put in place that would allow users to temporarily increase the sampling frequency for a specific application run or for a subset of nodes.

Telemetry Catalogue

Analogous to the data catalogues found in many data platforms, a telemetry catalogue allows users to search, browse, and curate (tagging and metadata management) telemetry data hosted in the storage layer. In order to better support computer-science research data-flows, a telemetry catalogue would allow users to publish telemetry data and make them accessible via a unique, global Digital Object Identifier (DOI). A DOI is an alphanumeric string assigned to uniquely identify a digital object and a de-facto standard in referencing experimental data sets in scientific publications. It is tied to a metadata description of the object as well as to a digital location, such as a URL, where all the details about the object are accessible. Allowing users to curate and publish data directly from the data catalogue aims at making a practical contribution towards a more research-friendly environment, one of the opportunities described in section 3.2.[2] The three main usage scenarios for a telemetry catalogue are:

1. Automatically catalogue the telemetry data associated with an application or experiment run: while the telemetry platform collects and holds the system and application telemetry generated during application execution, higher-level relationships between, e.g., different application runs might not be obvious, especially to other users. Metadata added via the telemetry catalogue adds a layer of interpretability to the data.

2. Find telemetry data based on metadata information: the data catalogue allows users to find telemetry based on terms specific to their application and their method of working.

3. Organize and publish experiment datasets related to a published analysis.

[2]This concept can be expanded so that not just the data but also the code that is developed as part of a research project can be published together as *Research Objects* [Bechhofer et al., 2010].

A data catalogue has not yet been implemented for SEASTAR, as it is less important for evaluating the overall platform architecture. Nevertheless, as the entry point to data discovery and curation, it is an important component for any production deployment of a telemetry platform. Several open-source data catalogue projects can potentially be used with SEASTAR, such as CKAN [CKAN, 2020] and Amundsen [Amundsen, 2020]. Both systems provide search and metadata management capabilities and provide plug-in mechanisms to connect them to different data sources. The difficulty with integrating an existing data catalogue solution is that their internal data models do not allow for an intuitive mapping and representation of a graph-based telemetry data model. One possible solution to that would be to resolve the explicit structure, interaction and allocation edges of the graph model into implicit, searchable metadata *tags*. This would however not solve the inherent time-variance of structure, interaction and allocations as constantly changing metadata tags would make the catalogue difficult to navigate. An alternative approach is to use the system only to catalogue static snapshots of system and application telemetry, i.e., the time seriesdata for specific experiments or application runs. Dynamic structure, interaction and allocation events could be flattened again into a separate (virtual) table which could then be used to reconstruct the spatial-temporal structure of platform and application if needed.

Privacy and Security

Opening up the telemetry data of an entire HPC platform to a broader audience of users and researchers undoubtedly opens up a broad spectrum of interesting opportunities. Especially machine learning-based resiliency and optimization systems would massively benefit from completely open access to telemetry data as it would give them much larger and more diverse training data sets to optimize their models. But while the benefits of completely open access to telemetry are significant, it always carries a certain risk for misuse and the associated privacy and security concerns.

Data misuse includes for example scanning telemetry data for sensitive information, such as usernames or passwords, other application-intrinsic information, or even personally identifiable information. Especially if the telemetry platform allows application users and developers to register custom, application-specific telemetry data (see section 2.1.2), it becomes very difficult to control whether sensitive data flows through the system. Furthermore, there are many areas of science in which research is invariably competitive so the exposure of information to rivals could have serious repercussions on careers and funding. Many model runs depend on and generate commercial-in-confidence data that must be protected.

Another, more sophisticated area of potential misuse are so-called side-channel attacks [Kocher, 1996] which apply statistical methods to the information gained from the operation of a computer system to infer protected information from other processes or users, such as cryptographic keys. Different classes of side-channel attacks use for example cache access data, data on the timing of computational operations, or data on the power consumption of hardware components, such as CPUs or cryptographic co-processors — all data that could potentially be collected and shared through a telemetry platform.

In order to mitigate these risks without completely abandoning the idea of open access to telemetry, we can apply threat modelling [Myagmar et al., 2005] as a structured approach to understand the different factors that can affect the security of an application or the overall system. Threat modelling provides and approach for identifying and assessing application threats and vulnerabilities, and then defining countermeasures to prevent or mitigate the effects of, threats to the system. If applied comprehensively, threat modelling can provide a clear "line of sight" across all areas of privacy and security concerns that justifies security efforts. The resulting threat model allows security decisions to be made rationally, with all necessary information available.

We expect that the assessment of security concerns and subsequent mitigation efforts will vary between different HPC platforms, based on parameters such as the composition of their user base (small and fixed vs. large and fluctuating groups) and the confidentiality of science workloads (unrestricted vs. classified). While we leave the creation of a detailed threat model for our telemetry platform architecture as future work, we will provide a list of possible counter-measures for future evaluation:

- **Preventing Ingestion** can be used to prevent sensitive data to enter the telemetry platform. This is the most secure but also most restrictive counter-measure.
- **Access Restriction** allows us to mark certain data sets that are known attack vector candidates as restricted. Users could still get access to the data, but only after explicitly requesting access.
- **Data Obfuscation** can be used to transform a sensitive data set into a non-sensitive data set. Techniques include masking out and filtering sensitive fields, or providing aggregates instead of individual data points.

Especially for implementing restricted access to data and possible data lineage and usage monitoring, the telemetry catalogue would provide a convenient platform to automate and provide transparency around these processes.

5.2.3 Processing and Querying

The processing and query layer provides scalable access to the structure and content of the distributed telemetry data stored across the different backends. Access to the semantic structure of platform and applications is provided directly via Neo4j's *Cypher* graph query interface. To access time seriesdata, SEASTAR provide three different modes: *stream*, *query*, and *batch*, each of them serving a specific set of use cases and analysis workflows based on their latency, complexity, and data volume requirements (see figure 5.5).

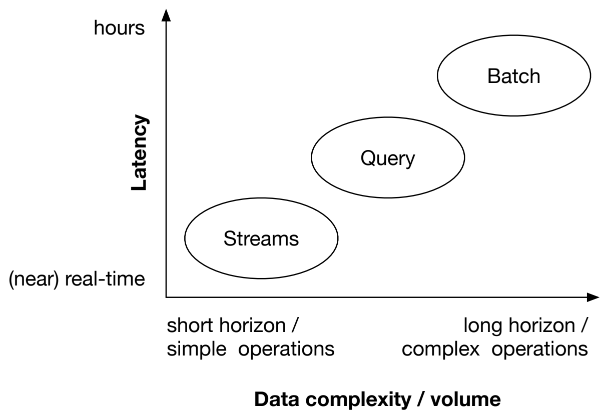


Figure 5.5: Streams, query, and batch are the three modes of access to telemetry data. Each of them serves a specific set of use cases based on their latency, complexity, and data volume requirements

At one end of the spectrum, stream queries provide near real-time access to current time seriesdata. The latency of stream queries is extremely low, but queries are limited to, depending on configuration, recent data and simple arithmetic and statistical operations. Stream queries are useful for direct monitoring tasks and simple application feedback loops that only rely on recent data. At the other end of the spectrum, batch processing is not limited on data size or complexity of operations but has much higher latencies in the order of minutes or even hours, depending on the volume and complexity of operations. This makes batch processing a good choice for more complex analysis tasks, such as in-depth analysis of historic telemetry data and machine learning tasks. Queries are situated somewhere between streams and batch. Queries are somewhat limited in terms of the data volumes and complexity of operations that can be processed efficiently but in return provide short query latencies in the order of seconds. This makes ad-hoc queries a good choice for use cases in which telemetry data is used to build simple application feedback loops, such as re-configuration of applications based on observed CPU, memory, or network consumption patterns as described in see section 2.2.2. SEASTAR provides four dedicated interfaces:

Graph Query: The graph query interface allows exploration of the semantic structure of platform and applications. Here we do not distinguish between different latency and data volume access patterns as the size of the semantic graph is relatively small compared to the time seriesdata. The telemetry graph can be queried directly via the native GDB APIs using the Cypher graph query language [Francis et al., 2018].

Time-Series Streaming: The time seriesstreaming interface allows real-time access to time seriesdata. This can be realized using the native TSDB APIs using its InfluxQL query language. Native time seriesqueries are very efficient, but allow only access to simple, precomputed operations, such as sum and average functions. Furthermore, storing data in a TSDB becomes inefficient and expensive at a certain point, so only a limited data horizon is available through this interface.

Time-Series Query: In order to reach further back in time, and to explore more complex relationships between telemetry data, the time seriesquery interface allows access to the full set of telemetry data stored in the telemetry data lake. Depending on the complexity of the query and the data involved, time seriesquery execution can take anywhere from seconds to minutes. Typically, time seriesqueries can be efficiently applied to data volumes in the order of hundreds of *megabytes*.

Time-Series Batch Processing: To overcome the limitations of time seriesqueries, the batch processing interface allows the deployment of custom processing functions directly on the storage layer, e.g., via the Spark or MapReduce programming frameworks. These frameworks allow for massively parallel execution and runtimes in the order of hours or even days. Batch processing has no limitations in terms of data volume and can be efficiently applied to data volumes in the order of *gigabytes* to *terabytes*.

Together, these four telemetry data access modes cover a broad set of data query and processing requirements. When implementing real-world telemetry-based workflows and application-support processes, architecture will often combine more than one query and processing technology: for example, and application support service could use real-time streaming to implement a short feedback cycle to make ad-hoc adjustments and decisions based on the currently observed data. The same application service could in addition use batch processing to implement longer feedback cycles that continuously re-train machine learning models that aid another aspect of application steering and configuration.

5.2.4 Analysis Workspaces

Analysis workspaces provide the main point for interactive interaction with telemetry data. They provide web-based data exploration and programming environments that are tightly integrated with the underlying storage, query and processing layers. SEASTAR integrates two popular web-based environments that are frequently used in data analysis and data science workflows: RStudio and JupyterLab[3]. JupyterLab [Granger and Grout, 2016] is a web-based interactive computational environment supporting multiple different programming languages, such as Julia, Python and R. JupyterLab uses the concept of a notebook document, which is a JSON document containing an ordered list of input/output cells containing code, text (using Markdown), mathematics, plots and rich media. RStudio [Allaire, 2012] is a web-based integrated development environment (IDE) for the R programming language. Figure 5.7 shows a side-by-side view of both environments.

Both systems are implemented as standalone HTTP servers that support multiple concurrent users and connections. While a single, multi-core server could potentially serve many RStudio and JupyterLab users, the unpredictable load of local computations spawned by users makes it difficult to provide predictable performance using the single-server model. Instead, SEASTAR provides a more flexible model in which new RStudio or JupyterLab server instances are spawned (and terminated) upon user request. This way, each user (or group of users) can have their own, dedicated server instance without causing side effects for other users (see figure 5.6). The provisioning of server instances is handled by the *Workspace Manager* service. The workspace manager, upon a users request, starts a preconfigured JupyterLab or RStudio container image from the *Workspace Image Registry* and returns the URL of the service via an HTTP proxy gateway back to the user. The analysis workspace containers have direct access to the graph and time seriesquery and processing APIs via a private network

[3]Both systems are found commonly in research computing ecosystems. See for example [Gandrud, 2013] and [Stubbs et al., 2020].

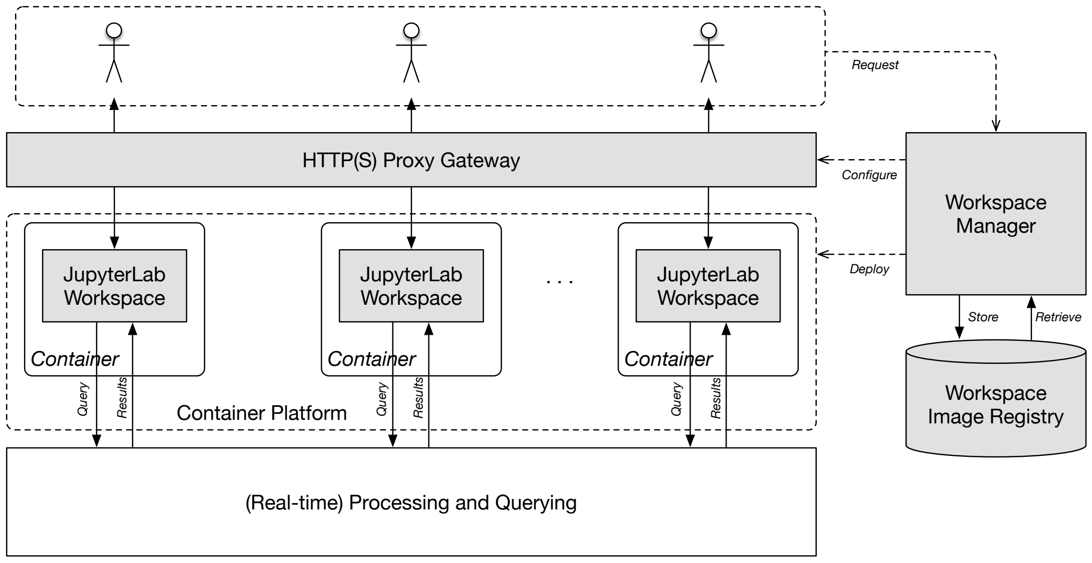
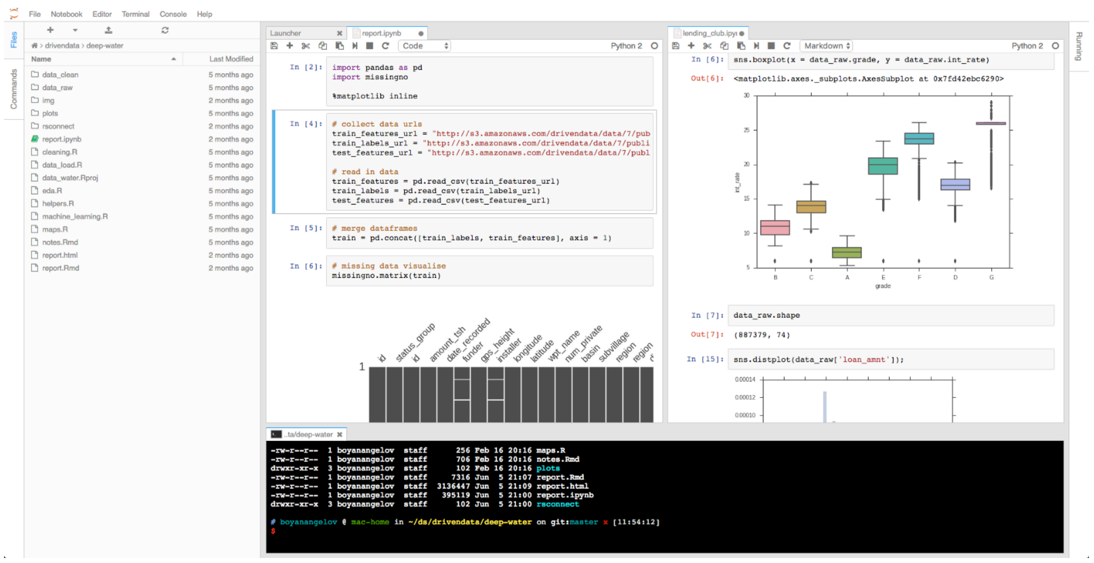


Figure 5.6: Workspaces are SEASTAR's ad-hoc deployable data exploration and programming environments. A central workspace manager starts preconfigured JupyterLab or RStudio container images from the workspace image registry.

connection.



drivendata > deep-water

Name	Last Modified
data_clean	5 months ago
data_raw	5 months ago
img	2 months ago
plots	5 months ago
rsconnect	2 months ago
report.ipynb	2 months ago
cleaning.R	5 months ago
data_load.R	5 months ago
data_water.Rproj	5 months ago
eda.R	5 months ago
helpers.R	5 months ago
machine_learning.R	5 months ago
maps.R	5 months ago
notes.Rmd	5 months ago
report.html	2 months ago
report.Rmd	2 months ago

```
In [2]:  import pandas as pd
         import missingno

         %matplotlib inline

In [4]:  # collect data urls
         train_features_url = "http://s3.amazonaws.com/drivendata/data/7/pub
         train_labels_url = "http://s3.amazonaws.com/drivendata/data/7/publi
         test_features_url = "http://s3.amazonaws.com/drivendata/data/7/publ

         # read in data
         train_features = pd.read_csv(train_features_url)
         train_labels = pd.read_csv(train_labels_url)
         test_features = pd.read_csv(test_features_url)

In [5]:  # merge dataframes
         train = pd.concat([train_labels, train_features], axis = 1)

In [6]:  # missing data visualise
         missingno.matrix(train)
```

```
In [6]:  sns.boxplot(x = data_raw.grade, y = data_raw.int_rate)

Out[6]:  <matplotlib.axes._subplots.AxesSubplot at 0x7fd42ebc6290>

In [7]:  data_raw.shape

Out[7]:  (887379, 74)

In [15]: sns.distplot(data_raw['loan_amnt']);
```

```
-rw-r--r--  1 boyanangelov  staff      256 Feb 16 20:16 maps.R
-rw-r--r--  1 boyanangelov  staff      706 Feb 16 20:16 notes.Rmd
drwxr-xr-x  3 boyanangelov  staff      102 Feb 16 20:16 plots
-rw-r--r--  1 boyanangelov  staff     7316 Jun  5 21:07 report.Rmd
-rw-r--r--  1 boyanangelov  staff  3136447 Jun  5 21:09 report.html
-rw-r--r--  1 boyanangelov  staff   395119 Jun  5 21:00 report.ipynb
drwxr-xr-x  3 boyanangelov  staff      102 Jun  5 21:00 rsconnect

# boyanangelov @ mac-home in ~/ds/drivendata/deep-water on git:master x [11:54:12]
$
```

Figure 5.7: Screenshot showing an interactive SEASTAR JupyterLab session running in a web browser.

5.2.5 Application Sidekicks

One of the main contributions of this work is to outline a concept to improve and simplify decoupled telemetry-driven HPC application architectures (see Section 3.2.2) and to reduce the overall complexity of HPC application design. Requirement *R.F10 Service Hosting* calls for a convenient way to host telemetry-based application support services, i.e., decoupled services that aid the execution of HPC applications. SEASTAR's application sidekick provide an implementation for this concept. For example, a machine learning model that takes a snapshot of an application's memory usage profile as input and classifies it as requiring adjustments or not would be implemented and deployed as an application sidekick.

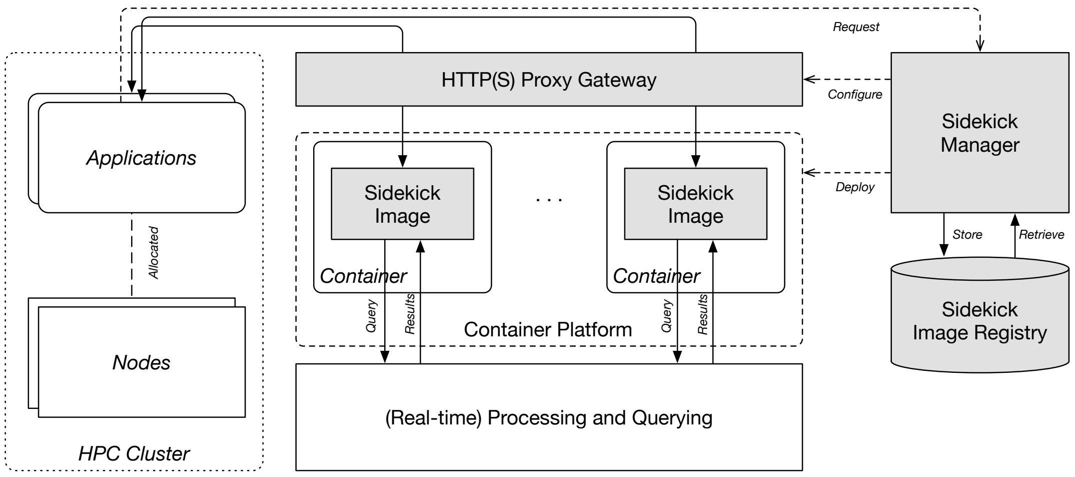


Figure 5.8: Application sidekicks are SEASTAR's implementation of application support services. A central sidekick manager deploys, starts, and stops user-created container images that are stored in a sidekick-image registry.

The architecture of the application sidekick service follows the pattern of the adhoc analysis workspaces. It consists of a central *sidekick-image Registry* which persistently stores the application sidekick-images, deployable application support services that expose their functionality via an API to one or more HPC applications. A *Sidekick Manager* is responsible for deploying, starting, and stopping these images upon request as Linux Containers (LXC). This can either be a manual invocation by the user or a programmatic request directly from an HPC application via the sidekick manager's REST API. application sidekick containers have direct access to the graph and time-series query and processing APIs via a private network connection. The service endpoints provided by the sidekick containers are made accessible to the HPC cluster and the applications running on it via an HTTP proxy.

Sidekick Image Catalogue

Collaboration and enabling an ecosystem of shared components and building blocks is an important aspect of the telemetry platform concept. Analogous to the telemetry catalogue, a sidekick catalogue allows users to share published sidekick images with each other. Once a version of a sidekick-image has been published to the sidekick registry, users can optionally and along with some metadata information that describes its purpose and scope, add it to the sidekick catalogue. This allows other users to investigate existing application sidekick images and gives them the option to re-use or re-purpose existing application support services instead of having to implement them from scratch.

5.3 Test Bed and Prototype

To evaluate the feasibility of the SEASTAR architecture, and to better understand how SEASTAR can be integrated with existing HPC systems and applications, we have built a prototype and testbed using AWS building blocks[4]. The testbed consists of two components within a single AWS Virtual Private Cloud isolated network segment (figure 5.9), a virtual HPC cluster and the SEASTAR implementation itself. We chose to build our own HPC cluster in AWS instead of using an existing, physical HPC cluster to reduce the complexity of our experimental setup to a level that is manageable within the scope of this work. Especially integration of SEASTAR with HPC system software, i.e., deploying telemetry collection agents and workload manager plug-ins would not be feasible on a production system. The SEASTAR prototype implements the architecture introduced in the previous section using. The prototype contains only a limited amount of custom software development — the majority of proof-of-concept telemetry platform functionality can be provided by combining existing technology, such as graph and time seriesdatabases, and AWS components. In this section, we describe the implementation architecture of our testbed, describe how SEASTAR and a “typical” HPC platform, in this case, a SLURM-based setup can be integrated, and provide some practical usage examples of the overall system.

[4]This work was supported through an *AWS in Education Research Grant* award from Amazon Web Services, Inc.

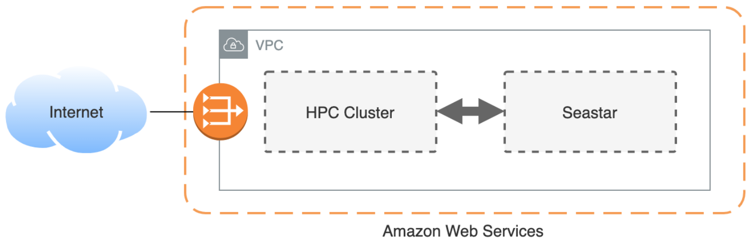


Figure 5.9: The testbed consists of a virtual HPC cluster and the SEASTAR prototype, both deployed within a single AWS Virtual Private Cloud network segment. An AWS NAT (Network Address Translation) Gateway provides connectivity to the internet.

5.3.1 Virtual HPC Cluster

In the context of this work, two requirements drive the AWS HPC Cluster implementation. Firstly, it must be possible to easily start and stop the cluster to control the cost of the experimental environment. And secondly, it must be possible to modify the system software, specifically the workload manager. We have chosen *CfnCluster* ("cloud formation cluster"), an open-source framework that deploys and maintains high-performance computing clusters on AWS. CfnCluster facilitates both quick start proof of concepts (POCs) and production deployments. It supports multiple workload managers, including SGE, Torque, and SLURM. We have chosen SLURM [Yoo et al., 2003] as the workload manager as it provides a plugin mechanism (PrEp API — *Pr*ologue and *Ep*ilogue API) that makes integration with SEASTAR easy to realize.

CfnCluster takes several configuration parameters, such as the AWS account ID, the AWS region in which the cluster should be deployed, the AWS EC2 instance types for compute and master nodes, networking and shared storage configuration. A *cfn-cluster* command-line tool takes the configuration and uses the AWS Cloud Formation APIs to bootstrap the cluster. Once bootstrapped, the command-line tool can be used to start, stop and reconfigure the HPC cluster on demand. Since we do not aim to run production workloads on the cluster, we chose relatively small AWS EC2 instances for the testbed configuration: we configured 64 `c5n.large` compute node instances which are designed specifically for HPC workloads due to their improved network throughput and packet rate performance and up to 100 Gbps network bandwidth. A single `c5n.large` instance provides two virtual CPU cores and 5.25 GB of memory, therefore the entire cluster provides 128 CPU cores and a total of 336 GB of memory.

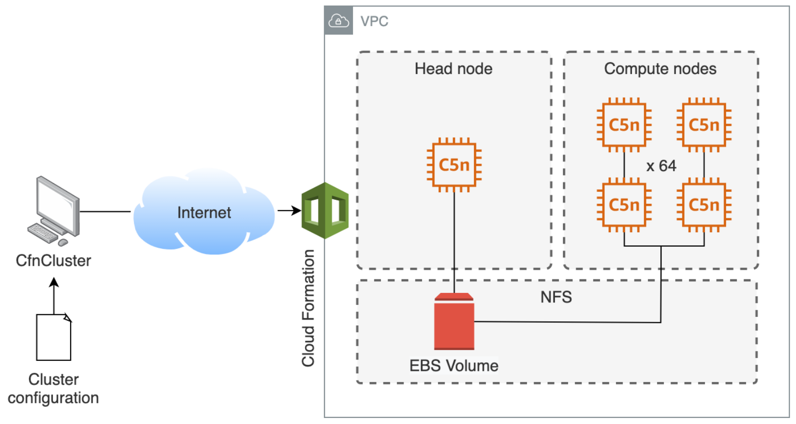


Figure 5.10: AWS deployment architecture of the HPC cluster testbed. The CfnCluster framework bootstraps and controls a fleet of AWS EC2 instances configured as a SLURM based HPC cluster.

An additional `c5n.large` AWS EC2 instance serves as the head node of the cluster running SLURM 20.02.6 as the workload manager. A shared 500 GB EBS (Elastic Block Storage) volume is mounted on the master node and shared via NFS to compute nodes. Figure 5.10 gives an overview of the deployment architecture.

5.3.2 SEASTAR Prototype

The purpose of the SEASTAR prototype is to evaluate the feasibility of the SEASTAR telemetry platform architecture, and to better understand how such a system can be integrated with existing HPC systems. Using AWS building blocks allow us to do this with little overhead: a lot of commercial data platforms are routinely implemented on AWS and a lot of complex and difficult to deploy and manage system components, such as data lake block storage, Linux container and Kafka clusters are available as SaaS components. This allows us to implement a functional SEASTAR prototype by combining and configuring components rather than developing them from scratch. Most of the software development went into the SEASTAR *Core Service* which is orchestrating SEASTAR's various components, the integration with the SLURM workload manager, which is discussed in the next section, and the SEASTAR Python library.

The implementation follows the architecture introduced in the previous section. As shown in figure 5.11, the multi-modal storage layer which is based on AWS S3 block storage, InfluxDB, and Neo4j is accessible via four different data query and processing interfaces: the native InfluxDB and Neo4j interfaces for real-time access, as well

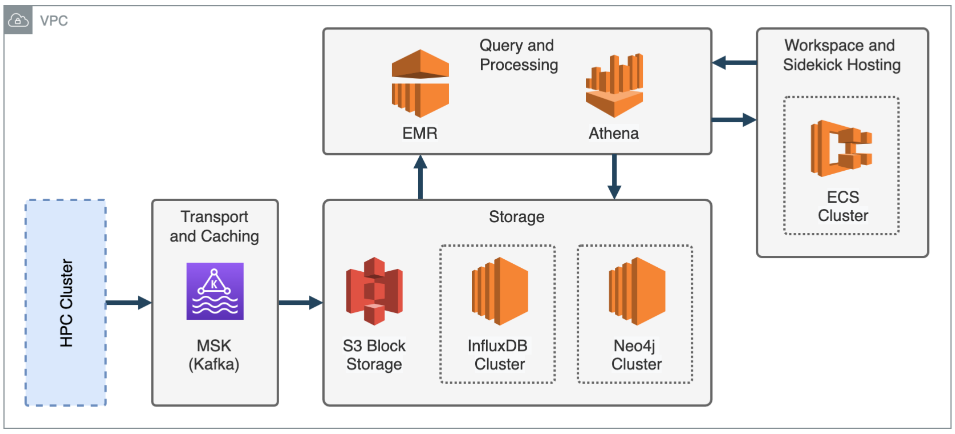


Figure 5.11: SEASTAR prototype AWS building blocks.

as AWS Athena and AWS EMR (Elastic Map Reduce) for data lake query and processing capabilities. AWS MSK (Managed Streaming for Kafka) provides data transport and caching, and AWS ECS (Elastic Container Services) provide the infrastructure for workspace and application sidekick instantiation. The remainder of this section described the individual component implementations in more detail.

Data Transport and Storage

The Kafka-based data transport and caching subsystem described in the previous section is implemented using AWS MSK (Managed Streaming for Kafka) (figure 5.13). AWS MSK is a fully managed Apache Kafka service that exposes the native Apache Kafka APIs without the overhead and complexity of managing a Kafka cluster in production. Using a managed Kafka services instead of other, AWS-specific data streaming services (e.g., AWS Kinesis) for which no equivalent open-source software exists, retains the flexibility to deploy SEASTAR components, such as the data agents and writers outside AWS, for example in an on-premise data centre. Like an on-premise installation of Apache Kafka, AWS MSK deploys multiple brokers that comprise the Kafka cluster. We have chosen to run three brokers on `m5.large` AWS EC2 instances. This number is based on a data ingress and egress rate of 2 MB/s each and a data retention time of 24 hours. Figure 5.12 provides some details on the number of brokers (based on EC2 instance type) as a function of data throughput requirements. This shows the scalability potential of the solution.

The SEASTAR storage layer consists of three distinct subsystems: the graph database, the time seriesdatabase, and the telemetry data lake. We use AWS EC2 virtual ma-

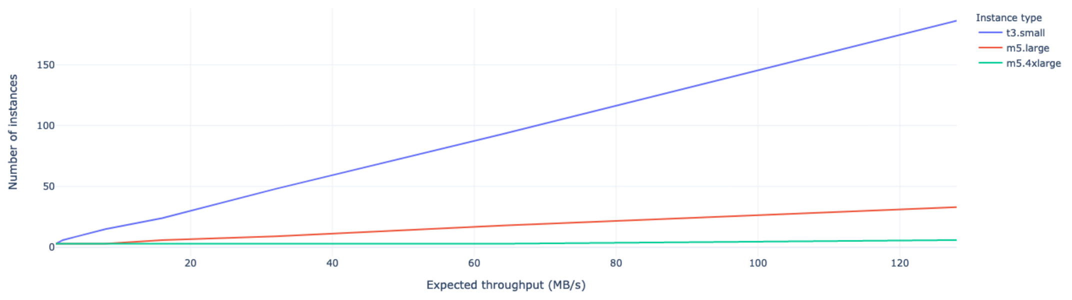


Figure 5.12: Diagram showing the suggested number of MSK Kafka brokers (based on EC2 instance type) as a function of data throughput requirements. With larger instance types, high throughput can be reached with smaller numbers of brokers.

chines to host Neo4j instances in a *Causal Clustering* configuration. Neo4j's causal clustering distinguished between *core servers* which are responsible for the long-term safekeeping of data and *read replicas* which are responsible for scaling out graph query workloads. While the scalability and fault-tolerance provided by causal clustering are not required for the scope of this prototype and experiments, we feel that it is still important to include this concept into the overall architecture to illustrate the scalability potential of the overall system. Neo4j is not an in-memory database, but multiple layers of RAM caching help to speed up the graph queries, hence it is important to find a good balance between instance disk I/O and memory size. AWS provides EC2 instances with locally attached NVMe SSDs to deliver high random I/O performance. For this prototype, we deploy three `c5d.2xlarge` instances with 16 GB of RAM, 200 GB NVMe SSD storage, and up to 10 Gbit/s network bandwidth. Details on Neo4j clustering and performance are discussed in [Raj, 2015] and [Holzschuher and Peinl, 2013].

For the time seriesdatabase, we use a similar setup of multiple AWS EC2 instances that host an InfluxDB high-availability cluster. The main motivation behind a clustered deployment is again to illustrate the scalability potential of the SEASTAR storage subsystem. InfluxDB distinguishes between data nodes and meta nodes. The meta nodes keep a consistent view of the metadata that describes the cluster, while data nodes are responsible for handling all writes and queries (figure 5.13). Optimal sizing is dependent on the database schema as well as on write and query load. We use the same `c5d.2xlarge` AWS EC2 instance types to deploy two data nodes. The meta nodes can run on modestly sized `t3.medium` AWS EC2 instances. We deploy three meta nodes as the cluster's consensus protocol requires a quorum to perform any operation, so there should always be an odd number of meta nodes.

The telemetry data lake storage is implemented on top of AWS' S3 object store and does not require any dedicated virtual servers. The AWS S3 web service interface provides access to a virtually limitless data storage facility. The basic storage units of AWS S3 are objects which are organized into so-called buckets. Each object is identified by a unique, user-assigned key and can be up to five terabytes in size with two kilobytes of metadata. For this prototype, we use a single AWS S3 bucket as the telemetry data lake, although future implementations might benefit from partitioning the data lake across multiple buckets. Data is written into S3, InfluxDB and Neo4j by several writers that read data from the Kafka stream. The writers are implemented in Python and use the Kafka, S3, InfluxDB, and Neo4j Python libraries to interface with source and target systems. They are deployed as headless services on multiple AWS EC2 instances that are wrapped in autoscaling groups which allow rule-based up- and down-scaling of the number of instances. While we do not actively use autoscaling in this prototype, it still illustrates how writer scalability can be achieved and matched against the scalability of the overall telemetry data pipeline.

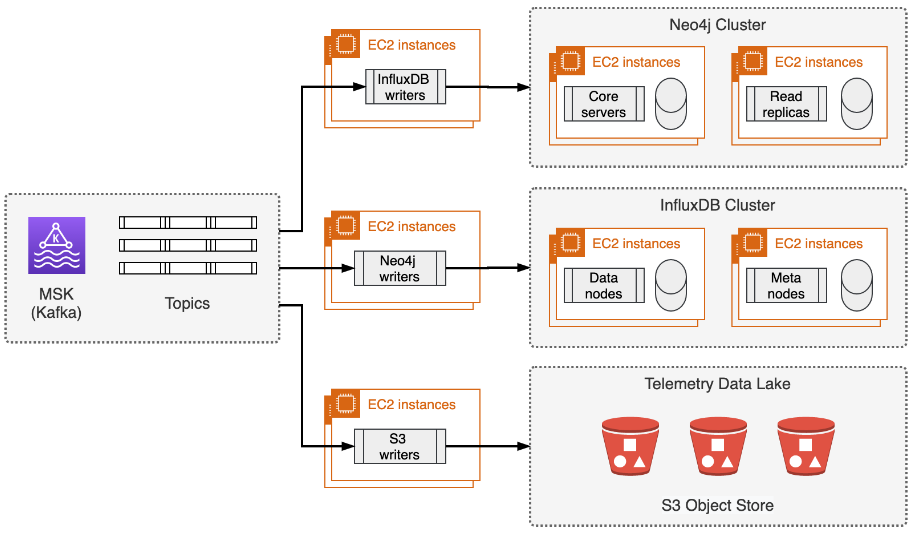


Figure 5.13: SEASTAR's data transport and storage implementation consists of a multi-modal storage layer based on AWS S3 block storage, InfluxDB, and Neo4j connected by an AWS MSK data pipeline to the node agents.

Query and Processing

Query and processing figure 5.14 follows the storage layer architecture. Cypher graph queries are handled directly by the Neo4j cluster. Neo4j uses the Bolt protocol, a connection-oriented network protocol that operates over TCP and WebSocket for client-server communication. All cypher client libraries implement the Bolt protocol transparently. An AWS Application Load Balancer (ALB) routes all query traffic between the Neo4j cluster and client libraries and can be used to centrally deploy user authentication (e.g., via OAuth2 or LDAP) and rate-limiting. The performance of the graph query interface directly depends on the configuration of the cluster which is described in the previous section. Analogously to graph queries, time-series queries are handled directly by the InfluxDB HTTP service endpoint exposed by the InfluxDB cluster. For both, graph and time seriesqueries, processing and storage are integrated within the same systems, which makes them sufficiently fast.

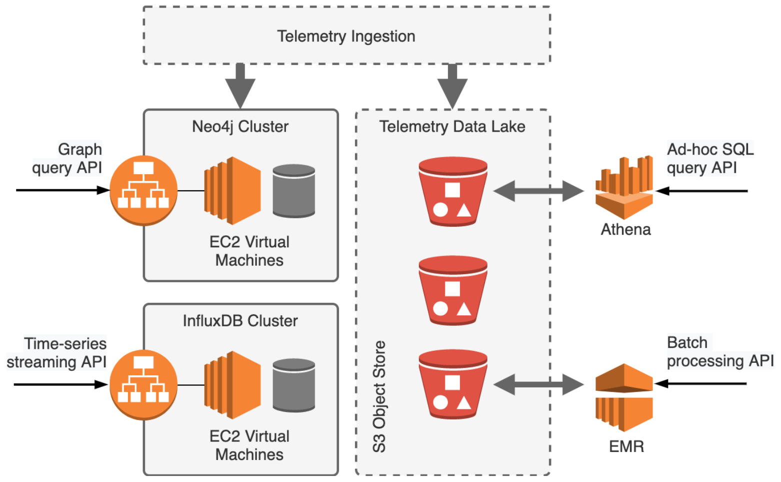


Figure 5.14: The SEASTAR prototype architecture provides four distinct query and processing interfaces using AWS components and services.

Ad-hoc SQL queries and batch processing are implemented differently. The telemetry data lake is implemented as an object store only, which means that it does not provide any integrated compute capacity. Moving data out of the data lake and into dedicated processing and query platform with the necessary compute capacity to run large-scale queries and processing workloads would be prohibitively expensive and slow. AWS solves this by providing several solutions that integrate directly with the S3 object storage, that do not require moving the data. Instead, compute capacity is brought close to the data residing in S3 through intelligent placement within the AWS infrastructure and data centres. We chose AWS Athena for providing large-scale query and AWS Elastic Map Reduce (EMR) for providing batch processing capabilities on top of the telemetry data lake (figure 5.14).

AWS Athena is an interactive query service that can access data directly in AWS S3 using standard SQL query syntax. Internally, Athena uses Presto [Sethi et al., 2019], an open-source, distributed SQL query engine and can scale automatically with the complexity and number of parallel queries. It supports data stored in different file formats, including the Apache Parquet columnar format which we have chosen as the internal data format of the telemetry data lake as it is supported by a broad spectrum of tools. Athena supports the standard Open Database Connectivity API (ODBC) which makes it accessible to a broad spectrum of tools, programming libraries and frameworks. Just like for the graph and time seriesquery endpoints, SEASTAR exposes the Athena ODBC interface via an ALB to the workspace and application sidekick layer.

AWS EMR is an implementation of Apache Hadoop, a collection of open-source software utilities that provide a software framework for distributed storage and data processing using the MapReduce programming model. In addition to the Hadoop Distributed Filesystem (HDFS) [Shvachko et al., 2010], AWS EMR provides the EMR File System (EMRFS), an implementation of HDFS that an Amazon EMR cluster can use for reading and writing data directly to and from AWS S3, which makes it an ideal candidate to provide processing capabilities on top of the telemetry data lake. Amazon EMR is deployed as a cluster, which is a collection of nodes running on AWS EC2 instances. AWS EMR can provision hundreds or even thousands of nodes to process data at any scale. The number of instances can be increased or decreased automatically using the AWS EC2 instance auto-scaling capabilities. EMR supports multiple distributed processing systems and frameworks on top of its MapReduce programming model. For this prototype, we use EMR with Apache Spark. Spark [Zaharia et al., 2016] utilizes in-memory caching, and optimized, distributed query execution for batch processing, interactive queries, real-time analytics, machine learning, and graph processing. It provides development libraries for the Java, Scala, Python and R programming languages. Since the EMR cluster is only needed when a batch processing job is submitted through the SEASTAR API, and data persistency is handled by the AWS S3 object store, we have configured EMR as an on-demand service. This means that the cluster nodes are deployed ad-hoc and terminated when the job has finished. In a production implementation of SEASTAR, this model can be expanded to per-user or user-group clusters, and usage-based cost accounting for batch processing of telemetry data. The interface to EMR is provided via the AWS API and allows users to start and stop the cluster and submit and control Spark jobs. Section 5.3.3 introduces a wrapper around this interface for a more intuitive integration with the SEASTAR programming library.

Workspace Implementation

The SEASTAR architecture suggests a Linux container-based implementation for user workspaces in order to isolate user environments and to provide predictable performance. AWS provides two container deployment services: Elastic Kubernetes Service (EKS) and Elastic Container Service (ECS). For this prototype implementation, we have chosen ECS as it allows us to run containers (via the AWS Fargate engine) without managing the underlying compute infrastructure (figure 5.15).

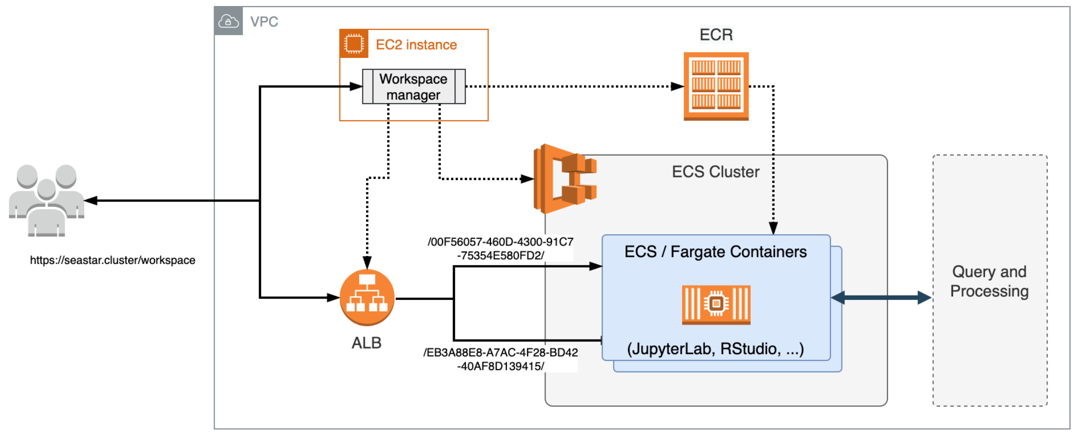

VPC
EC2 instance
Workspace manager
ECR
ECS Cluster
https://seastar.cluster/workspace
/00F56057-460D-4300-91C7 -75354E580FD2/
ECS / Fargate Containers
Query and Processing
ALB
/EB3A88E8-A7AC-4F28-BD42 -40AF8D139415/
(JupyterLab, RStudio, ...)

Figure 5.15: The workspace implementation architecture uses an AWS Elastic Container Service (ECS) cluster, preconfigured workspace images in an Elastic Container Registry (ECR), and an Application Load Balancer (ALB) managed and controlled by the SEASTAR workspace manager service.

The ECS/Fargate container cluster is situated within the same private network segment (VPC) as the other SEASTAR components, so it has access to all interfaces exposed in the query and processing layer. We have preconfigured a workspace container image with JupyterLab that has the SEASTAR programming library (see section 5.3.3) and dependent libraries and ODBC drivers that are required to connect to the query and processing interfaces installed and configured. The container image is stored in AWS Elastic Container Registry (ECR), a managed Docker container registry that allows us to store, manage, share, and deploy container images in AWS ECS. The workspace manager, a service and API written in Python is deployed on a single AWS EC2 instance and manages the workspace on behalf of a user. The API functions provided by the workspace manager allow users to deploy, start, stop, and destroy a workspace. When a new workspace is requested, the workload manager launches the requested workspace image (in this case JupyterLab) on ECS/Fargate, attaches the port of the container's web interface to a route on an application load balancer, and returns the unique URL under which the workspace can be reached.

Application-Sidekick Implementation

The environment to deploy application sidekicks is similar to the workspace environment and builds on the same concepts and technology. Instead of the preconfigured container images deployed in the case of workspaces, the container images deployed as application sidekicks are developed and uploaded to the container registry by the users. The basic mechanics here are the same: a SEASTAR sidekick manager, deployed on a single AWS EC2 instance, controls container deployment on an ECS cluster and routes and exposes the interfaces exposed by the containers as a unique URL through an application load balancer (figure 5.16).

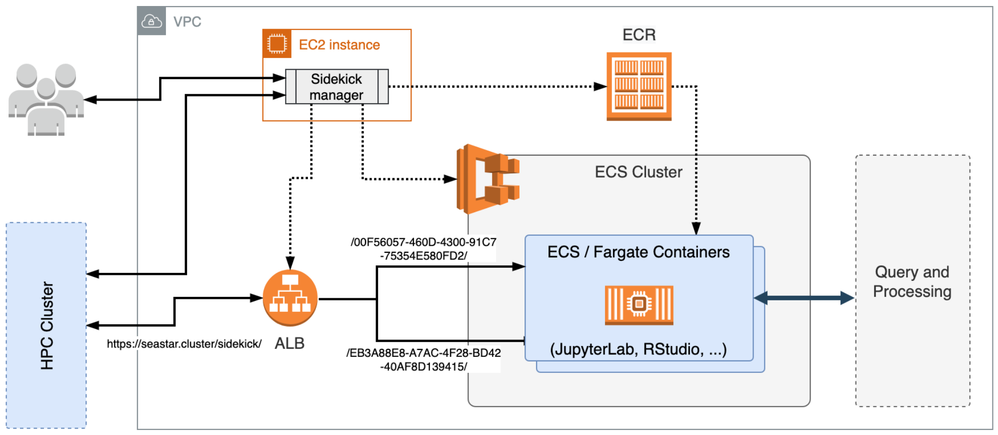

VPC
EC2 instance
Sidekick manager
ECR
ECS Cluster
HPC Cluster
/00F56057-460D-4300-91C7 -75354E580FD2/
ECS / Fargate Containers
(JupyterLab, RStudio, ...)
Query and Processing
https://seastar.cluster/sidekick/
ALB
/EB3A88E8-A7AC-4F28-BD42 -40AF8D139415/

Figure 5.16: The application sidekick implementation architecture uses an AWS Elastic Container Service (ECS) cluster to deploy user-created container images and an Application Load Balancer (ALB) that exposes the sidekick interfaces to the HPC jobs. A sidekick manager service manages and controls sidekick creation and lifecycle.

Just like interactive workspaces, application sidekicks need access to the interfaces exposed by SEASTAR's query and processing layer. For convenience, we built a sidekick *base image* with the SEASTAR programming library and dependent libraries and drivers installed and preconfigured. By itself, the base image does not provide any functionality, but user-built images can *inherit* from it to simplify the sidekick development process. The API functions provided by the sidekick manager extends the workspace manager API with functions to upload sidekick container images to the container registry. API functions to mark images as *shared* and to list all shared images provide a foundation for future implementations of an application sidekick catalogue as outlined in the previous section.

To add additional guidance and convenience for sidekick developers, future versions of SEASTAR should consider providing several specific sidekick template images that aim towards specific use case patterns. An example for this would be a template image providing a pre-built REST or WebSocket service that can be easily modified or extended by the developer, without having to be concerned with the intricacies.

5.3.3 SEASTAR Programming Library

A programming library can help to make the SEASTAR data platform intuitively usable for users and developers by introducing high-level concepts and abstraction of the platform in a consistent, programmatic way. This approach is sometimes called an *opinionated library* as it imposes the creator's view of concepts, abstractions and design-patterns on the developers. While an opinionated library constrains the degrees of freedom of how a developer can interact with the underlying system, it also reduces cognitive load, which is desirable if a broad spectrum of users is targeted. As part of the prototype, we have sketched out the first version of a SEASTAR programming library. The library consists of two independent interfaces: the SEASTAR platform API and the telemetry access API. The platform API provides functionality for controlling the user-facing capabilities of the platform. This includes:

- *Telemetry asset management*: management and curation of telemetry data assets, managing metadata, allocation and control of telemetry platform resources. The implementation of this interface integrates with the *Platform Manager* service.
- *Workspace management*: creation, destruction, and control of user workspaces. The implementation of this interface integrates with the *Workspace Manager* service (see section 5.2.4).
- *Sidekick management*: creation, destruction, and control of user-developed application sidekicks. The implementation of this interface integrates with the

Sidekick Manager service (see section 5.2.5).

Listing 16 shows example invocations of the platform API. It is targeted at programmatic interaction with SEASTAR, either directly by the user, indirectly by a user's HPC job which can for example use the API to start a required sidekick service, or through integration with other services, such as workload managers (see section 5.3.4). It can also serve as the integration layer for a future graphical user interface or web portal.

```python
from seastar import platform as p

# Connect to the Seastar platform manager
sp = p.PlatformClient(access_key="XXXXXXXXXXXXX", secret_key="XXXXXXXXXXXXX")

# Create and launch a new workspace
wsc = p.WorkspaceConfig(type="JupyterLab", num_cpus=4, memory=16)
ws = sp.create_workspace(wsc)
ws.start()
print("Workspace available at: " + str(ws.url)

# Re-connect to a workspace and shut it down
ws2 = sp.get_workspace(ws.url)
ws2.shutdown()
```

Listing 16: SEASTAR API example showing how to create and manage workspaces and application sidekicks.

While the platform API provides access to and control of telemetry platform components, the telemetry access API allows for the interaction with the telemetry data that is stored in the platform. Its main function is to provide a common access layer to the four different query and processing interfaces (see section 5.2.3). The native Python libraries for GraphQL, InfluxDB, AWS Athena's ODBC interface, and the AWS library to submit and control Spark jobs to an EMR cluster all follow different approaches. The telemetry access API consolidates them in a lightweight abstraction layer combined with a single authentication and authorization mechanism. While the native capabilities of the interfaces are retained, it gives the users a more integrated experience of the telemetry platform. Furthermore, the telemetry access API is designed in a way that allows integration of the query and processing APIs with the telemetry access building blocks managed by SEASTAR: concepts like applications and jobs can be referenced directly in the telemetry access API as it is illustrated in listing 17.

The SEASTAR programming library that we have developed as part of this prototype provides the basic functionality required to experiment with the platform and to illustrate a path towards coherent and transparent programmatic interaction with the complexities of a distributed data platform. There are many topics for further improvements and research, for example integrating graph- and time seriesqueries into a single query interface. We discuss this further in the future work section (section 6.3).

```python
from seastar import platform as p

# Connect to the Seastar platform manager
sp = p.PlatformClient(access_key="XXXXXXXXXXXXX", secret_key="XXXXXXXXXXXXX")

# Get a telemetry graph handle and run a Cypher query
tg = sp.telemetry_graph()
result = tg.query(
    'MATCH (node1:CNode {hostname: "node01"})-[:ALLOCATED_TO]-(proc:Process)'
    'WHERE proc.destroyed = 0'
    'RETURN proc')
print("Result: {0}".format(result))

# Get a time seriesdatabase handle and run an InlfuxQL query
td = sp.timeseries_data()
result = td.query(
    'SELECT current FROM cpu_frequency'
    'WHERE "hostname" = "node01"')
print("Result: {0}".format(result))

# Get a handle to the EMR cluster and launch a SPARK job
emr = sp.emr_service()
job = emr.upload_job("./AnomalyTrainV01.py")
emr.run(type="spark", name="AnomalyTrain", job=job,
        num_executors=5, executor_cores=5, executor_memory="20g")
```

Listing 17: SEASTAR API example showing how to query the telemetry graph, telemetry time seriesdata and submitting a Spark batch processing job.

5.3.4 Workload Manager Integration

As we have seen in the previous section, the SEASTAR programming library provides programmatic access to both telemetry platform capabilities and the telemetry data itself. However, interfacing via the programming library requires explicit amendment of existing user programs, scripts and workflows. For example, in order to allocate telemetry platform resources, register a new application run in the data catalogue, or starting an application sidekick service before execution would require the developers to add SEASTAR-specific code to their application. Integrating core SEASTAR functions with an HPC cluster's workload manager provides an alternative approach, allowing unmodified and even telemetry-agnostic applications to take advantage of the capabilities of a telemetry platform.

Many, if not all multi-tenant HPC clusters are controlled by a workload manager. A workload manager provides the mechanism through which user applications are launched on the cluster. It has three key functions:

- Allocate exclusive and/or non-exclusive access to compute nodes to users for a user-defined duration of time.
- Provide a job-control framework for starting, executing, and monitoring jobs on the set of allocated nodes.
- Arbitrate contention for resources by managing a queue of pending jobs.

Like many other workload managers, the SLURM workload manager that we use on our virtual HPC cluster is controlled via a set of command-line tools to control and monitor jobs and job scripts that describe the application a user wants to run and its resource requirements. SLURM job scrips resemble simple UNIX shell scripts with a declarative preamble that contains properties such as the number of nodes, CPUs, and memory required, and the duration for which these resources are required. SLURM provides a flexible plugin mechanism that allows us to customize the workload manager's behaviour and extend or modify its functionality. We have developed a custom SLURM plugin-in that allows users to declare the following SEASTAR-specific functionality as part of a job script:

- Attach metadata as key-value-pairs to the telemetry data that is collected during job execution via the `SS_METADATA_TAGS` keyword.
- Start an application sidekick service before executing the actual job via the `SS_SIDEKICK` keyword.
- Associate a sidekick WebSocket signal to a job command, for example `SUSPEND` via the `SS_BIND_SIGNAL`.

```bash
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --time=01:00:00
#
#SS_METADATA_TAGS {version:"1.1", iteration:"25", project:"testbed"}
#SS_SIDEKICK      {id:"ocw/watchdog:1.16"}
#SS_BIND_SIGNAL   {endpoint:"ws:/ocw/watchdog:1.16/signal", \
#                  signal:"ABORT_SUGGESTED", state:"CHECKPOINT+CANCEL"}

# Start the job running using OpenMPI's "mpirun" job launcher
mpirun ./my_application
```

Listing 18: Example SLURM job script that uses SEASTAR-specific declarations in the preamble to instantiate and integrated with an application sidekick. The application in this example is not aware of SEASTAR – integration happens purely at the workload-manager-level.

Listing 18 shows a SLURM script that uses SEASTAR-specific declarations in the preamble. In this example, an application sidekick service is requested to monitor the execution of a user's application. In this case, the sidekick provides a single signal `ABORT_SUGGESTED` via a WebSocket connection. The signal is then associated with SLURM's checkpoint feature which is intended to save a job state to disk as a checkpoint and terminate the execution. Our cluster testbed uses the Berkeley Lab Checkpoint/Restart (BLCR) library [Hargrove and Duell, 2006] which is installed on the cluster nodes. The MPI application `my_application` that is executed in this example is completely agnostic of the sidekick. The feedback-control-loop is established between SEASTAR and the SLURM job-control service. While this mode of integration is limited to very simple feedback-control-loops, it allows existing HPC applications to take advantage of telemetry-based job-control without any additional development effort. This can help HPC providers to reduce premature job terminations and wasted compute cycles, while at the same time increase resiliency for a broad spectrum of existing applications.

The sequence diagram in figure 5.17 illustrates the interaction between SLURM and SEASTAR. The SLURM plug-in interfaces with the platform API to register the job and its metadata with the SEASTAR Platform Manager and to start and terminate application sidekick services. SLURM's internal plug-in APIs are used to carry out job-control functions. The details of managing authentication and sidekick service

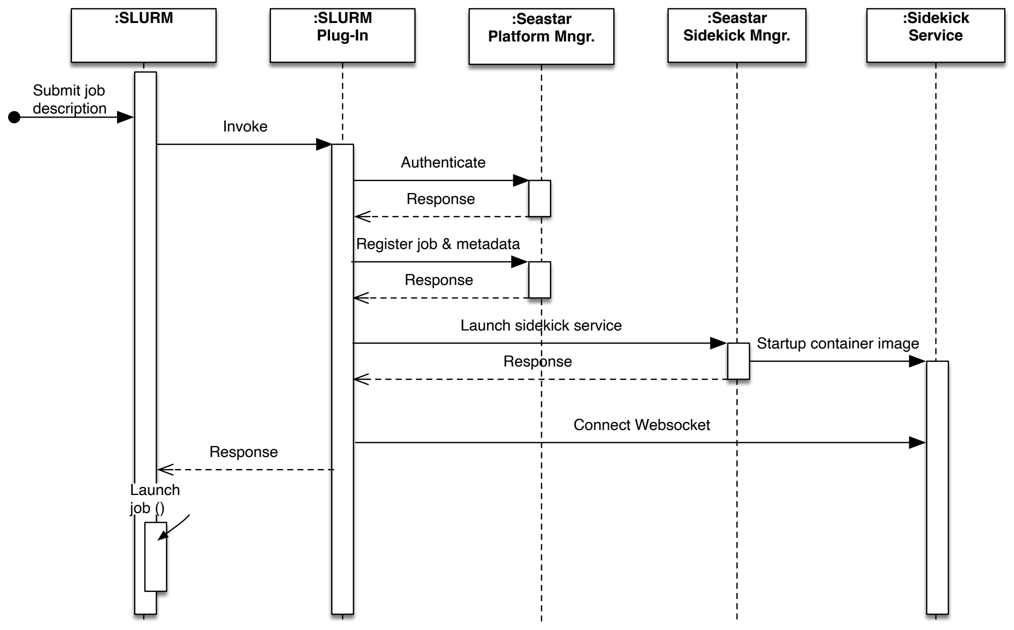


Figure 5.17: Sequence diagram illustrating the interaction between SLURM, SEASTAR and application sidekick service via a SLURM plug-in.

URLs are handled transparently within the plug-in. Job submission starts with the users submitting a job description to SLURM which invokes the plug-in. After parsing the job description for relevant information, the plug-in contacts the SEASTAR *Platform Manager* to authenticate and register the job and its metadata. In the next step, the plug-in contacts the SEASTAR *Sidekick Manager* to initiate the launching of the requested sidekick service. The *Sidekick Manager* launches the corresponding container image and returns the service access URL to the plug-in which can then establish a WebSocket connection with the sidekick service. Once this sequence is complete and the connection is established, SLURM launches the job. The plug-in continues to listen to the WebSocket connection for a relevant signal in the background and invokes the SLURM job-control if a signal matches the one declared in the job description.

5.4 Use-Case Demonstration

In this section, we demonstrate how the telemetry platform concept can be applied to different HPC application use cases. The focus of this demonstration is not to validate and benchmark the SEASTAR prototype implementation but to explore how the telemetry platform concepts, implemented by SEASTAR, support a more streamlined development workflow and support more flexible decoupled HPC system architectures, and easier development of more generic resilience and optimization capabilities for HPC applications. We have picked an example that we previously discussed in Section 2.2 and implemented it in our testbed: a machine learning-based application anomaly detection service that can be trained to detect common application failure patterns. We picked this particular example because it covers a broad spectrum of telemetry platform capabilities and represents one of the key opportunities — supporting architectures utilizing machine learning techniques — that we have identified in section 3.2. In this section we provide an overview of the implementation architecture, the development workflow, as well as the experimental setup, results, and evaluation.

5.4.1 ML-Based Application Anomaly Detection

One of the key capabilities of SEASTAR is that it enables large-scale processing on a large corpus of historic telemetry data. Together with the application sidekick concept, this allows for the development of machine learning-based feedback-control-loops that use the telemetry data lake and batch processing facilities to continuously train and retrain machine learning models on telemetry data and the sidekick services to run online classifiers of live telemetry streams against the model. Without an integrated telemetry platform, this type of feedback-control-loop would be very difficult to implement from an application developer's perspective as the data and processing infrastructure required is complex to set up and the shielded network environments in which HPC systems run would make it difficult or even impossible to integrate.

As part of the evaluation of our prototype, we have chosen to implement an exemplary machine learning-based application anomaly detection system to illustrate the significant reduction in implementation complexity with a platform like SEASTAR. Anomaly detection [Chandola et al., 2009] is the identification of rare observations which raise suspicions by differing significantly from the majority of the data. It is a common technique in large-scale network intrusion detection [Bhuyan et al., 2013] and fraud detection in the financial domain [Ahmed et al., 2016]. It has been suc-

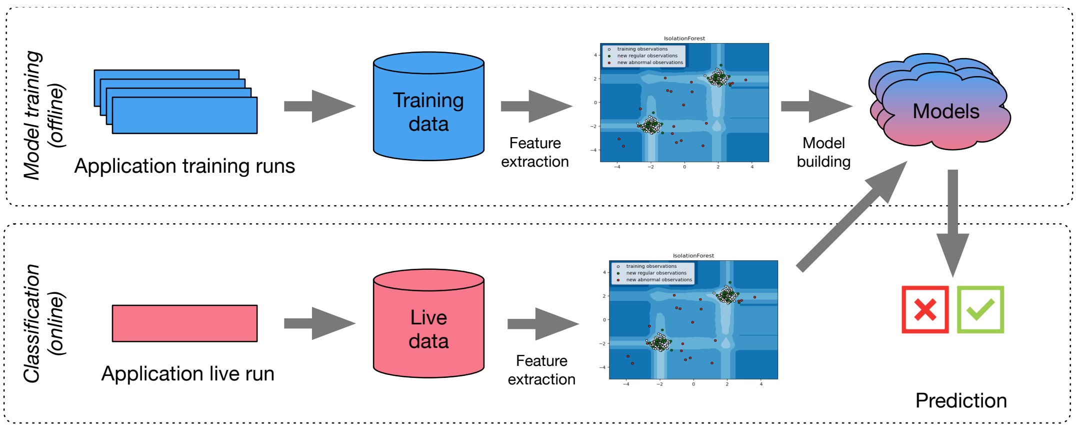


Figure 5.18: Machine learning flow for application anomaly detection. Machine learning models built using (synthetically generated) historical telemetry data are use for classifying runtime telemetry data.

cessfully applied both to HPC application anomaly detection [Tuncer et al., 2017b] and HPC system hardware anomaly detection [Borghesi et al., 2019]. Our use case is based on Borghesi's work and adds a system and data architecture perspective that has not been proposed previously. For the practical evaluation, we follow the experimental setup and anomaly detection approach presented in [Tuncer et al., 2017b]. The overall approach is shown in figure 5.18[5]. We collect application telemetry through a series of application runs, both with injected synthetic anomalies and without. Based on the training data, we extract several statistical features and label the data with the type of anomaly introduced. Using the labels and features, we train a non-parametric supervised machine learning model using *Decision Trees* (DTs) and *Random Forest* classifiers. Both methods are effective in previous work. We have chosen these two specific classifiers as they have been reported to be of high prediction accuracy in previous work.

Implementation

The implementation (figure 5.19) consists of two main components: a *Model training* component that uses the batch processing facilities of the telemetry data lake to implement model training, and a *Prediction service* component build as an application sidekick service. Our implementation uses the SLURM plug-in introduced in the

[5]we focus on a single application in this evaluation, but the same approach and implementation can easily be expanded to a broad spectrum of different applications.

previous section to implement the feedback-control-loop between the anomaly detection service and the applications: if the anomaly detection service suggests that an application exhibits problematic behaviour, the application is simply terminated by the workload manager. Much more sophisticated control loops could be implemented in which individual applications react differently on a signal from the anomaly detection service, but this would not add much additional value and insights to this use case as it focuses on ML-based feedback-control-loops.

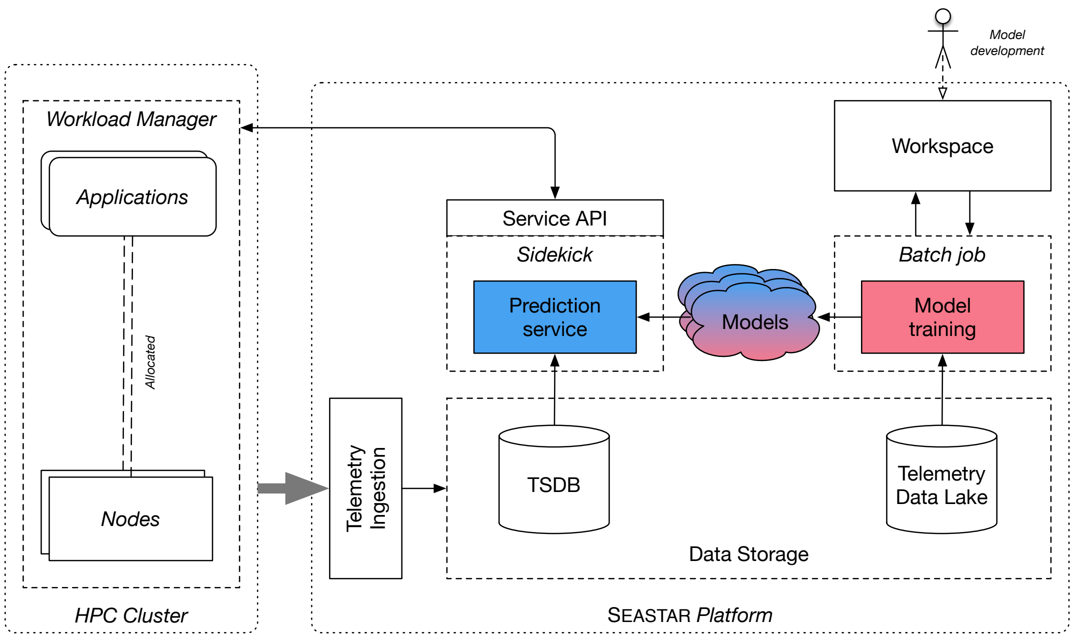


Figure 5.19: Implementation of an ML-based application anomaly detection service.

The **model training** component implements a Decision Tree (DT) and a Random Forest (RF) classifier. DT creates a model that predicts the value of a target variable by learning simple decision rules, represented as a tree, inferred from the data features. The tree can be seen as a piecewise constant approximation. RF is an ensemble learning method that fits several decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The model training component is implemented in Python and uses the Scikit-learn [Pedregosa et al., 2011] programming framework for interactive data exploration and PySpark ML for full-scale, parallelized model training. Scikit-learn provides a broad spectrum of classification, regression and clustering algorithms, including support vector machines, random forests, gradient boosting, and k-means. PySpark ML [Lovrić et al., 2019] is a Python wrapper for the Spark Machine Learning Library (MLib) [Meng et al., 2016]. It supports a similar feature set as Scikit-learn, but can parallelize the data processing and model training operations using the underlying Spark framework. The trained models are written to a project-specific location in the telemetry data lake using the training data, which is also preserved in the telemetry data lake. Telemetry data to be included in the training and re-training of the machine learning model is tagged accordingly using the SEASTAR *Platform Manager* via the SLURM plug-in (see section 5.3.4). A preprocessing step collects and prepares the data for the training step. An interactive workspace can be used to develop and continuously evaluate the models that are serialized and stored in a project-specific location in the telemetry data lake using the Open Neural Network Exchange (ONNX) file format, an open standard for machine learning interoperability [Foundation, 2021]. Once stored, the models can be accessed and loaded by the prediction service.

The **prediction service** is the application-facing component of the anomaly detection architecture. It is implemented in Python as an application sidekick service and exposes a WebSocket API that emits events based on the prediction outcome of online telemetry data. For deployment and usage, the service code is then packaged into a docker container and uploaded to and registered with SEASTAR's sidekick registry. Just as the model training component, the prediction service uses the Scikit-learn programming framework to carry out the prediction task. Since the prediction is much less computationally intensive, we do not need to parallelize it or run it as a SEASTAR batch processing job. The ONNX serialization format ensures that a production machine learning model created by PySpark ML can be loaded into Scikit-learn. Once the model has been loaded, connects to the time seriesstreaming API to gather sample data from the application it has been set up for a preconfigured amount of time (epoch length). Once the data has been gathered, Scikit-learn is used to predict the class of anomaly (or healthy) of the application based on the sample set. The result of the prediction is emitted as an event via the WebSocket API. The collection and prediction step is repeated until no more application telemetry is available through the time seriesstreaming API, i.e., the application has terminated.

Development Workflow

Important for the overall usability and developer-friendliness, and, consequentially for the uptake of a telemetry platform is the development workflow or developer *experience* it provides. A development workflow ideally supports developers or researchers along the entire application lifecycle, which means iteratively building, testing, deploying, and running the software they are developing. The development workflow for the application anomaly detection service in SEASTAR is depicted in figure 5.20.

Every new development, whether it is a simple data analysis or a complex application support service, starts with creating a new project. A project in SEASTAR has multiple purposes: firstly, it provides an exclusive storage area in the telemetry data lake for user-created and derived datasets, and for sidekick-images in the image registry. Secondly, it allows an HPC service provider to track and potentially bill for resource usage. Multiple users can be *members* of a project, which makes it easy to collaborate, and share resources and artefacts.

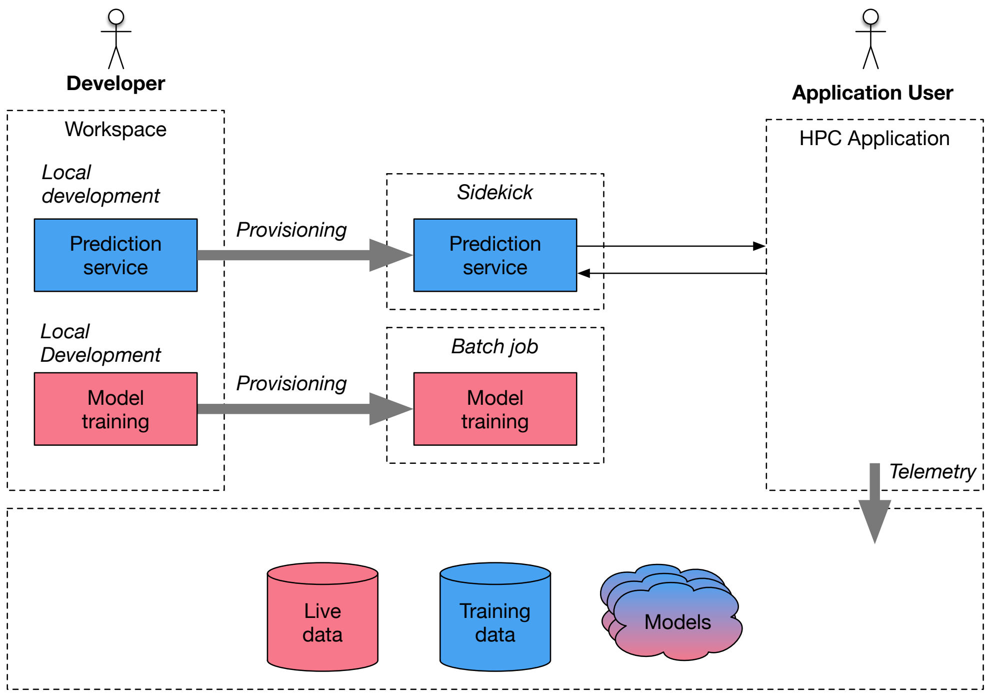
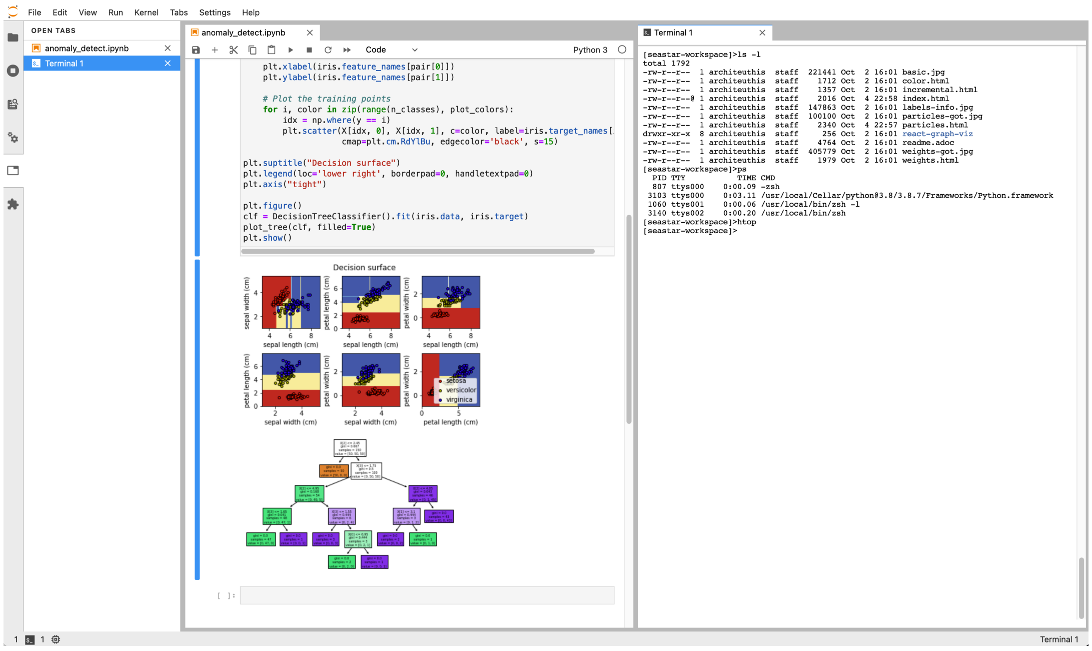


Figure 5.20: Development workflow for the application anomaly detection service: workspaces play a central role for the local development and provisioning of application sidekicks and batch jobs. Service development happens directly on the SEASTAR platform, with access to live data. Transferring telemetry data to a local development environment can therefore be avoided.

Once the project has been created, three main development tasks need to be accomplished: (1) collecting the sample data, (2) developing, building, testing, and deploying the model training component, and (3) developing, building, testing, and deploying the prediction service. To accomplish the first task, a series of jobs, in this case, synthetic jobs, are run on the HPC cluster and tagged with the project ID and the metadata required to train the model. This is done by adding the required information to the preamble of the SLURM job script (see section 5.3.4). For building the model training component, the first step is to start a new JupyterLab workspace. The workspace gives the developer an interactive development environment with full access to the training data (figure 5.21). Downloading telemetry data to a local development environment can hence be omitted.



```
    plt.xlabel(iris.feature_names[pair[0]])
    plt.ylabel(iris.feature_names[pair[1]])

    # Plot the training points
    for i, color in zip(range(n_classes), plot_colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, label=iris.target_names[
                    cmap=plt.cm.RdYlBu, edgecolor='black', s=15)

plt.suptitle("Decision surface")
plt.legend(loc='lower right', borderpad=0, handletextpad=0)
plt.axis("tight")

plt.figure()
clf = DecisionTreeClassifier().fit(iris.data, iris.target)
plot_tree(clf, filled=True)
plt.show()
```

```
[seastar-workspace]>ls -l
total 1792
-rw-r--r--  1 architeuthis  staff  221441 Oct  2 16:01 basic.jpg
-rw-r--r--  1 architeuthis  staff    1712 Oct  2 16:01 color.html
-rw-r--r--  1 architeuthis  staff    1357 Oct  2 16:01 incremental.html
-rw-r--r--@ 1 architeuthis  staff    2016 Oct  4 22:58 index.html
-rw-r--r--  1 architeuthis  staff  147863 Oct  2 16:01 labels-info.jpg
-rw-r--r--  1 architeuthis  staff  100100 Oct  2 16:01 particles-got.jpg
-rw-r--r--  1 architeuthis  staff    2340 Oct  4 22:57 particles.html
drwxr-xr-x  8 architeuthis  staff     256 Oct  2 16:01 react-graph-viz
-rw-r--r--  1 architeuthis  staff    4764 Oct  2 16:01 readme.adoc
-rw-r--r--  1 architeuthis  staff  405779 Oct  2 16:01 weights-got.jpg
-rw-r--r--  1 architeuthis  staff    1979 Oct  2 16:01 weights.html
[seastar-workspace]>ps
  PID TTY           TIME CMD
  807 ttys000    0:00.09 -zsh
 3103 ttys000    0:03.11 /usr/local/Cellar/python@3.8/3.8.7/Frameworks/Python.framework
 1060 ttys001    0:00.06 /usr/local/bin/zsh -l
 3140 ttys002    0:00.20 /usr/local/bin/zsh
[seastar-workspace]>htop
[seastar-workspace]>
```

Figure 5.21: Screenshot showing a Scikit-learn model training and evaluation session side-by-side with an interactive terminal in a JupyterLab development workspace.

Within the workspace, we can now start building the data preparation and training model code interactively. While the performance of a workspace is limited, it is sufficient to build and train models with a subset of the data. Once the approach has been verified, we can wrap the preparation and training steps into a PySpark job that can run on AWS EMR. We use the SEASTAR platform manager API to launch an EMR cluster, also from within the JupyterLab workspace. In this development workflow, the workspace becomes the central interaction point for the developers, both for interactive exploration of the telemetry data, and for controlling the compute-intensive model training EMR jobs. The benefit of this approach is that code developed in a tightly integrated SEASTAR workspace is both shareable with other developers on the platform but also does not require any specific environment configurations or setup instructions.

Similarly, the prediction service can also be developed locally in the workspace and then wrapped into an application sidekick service that can be deployed via the SEASTAR API. Interactive evaluation of the prediction methodology can be performed directly in the workspace, using the time seriesstreaming API to feed data directly into the Scikit-learn prediction functions. Besides the interactive Python notebooks, a JupyterLab workspace has provided access to the underlying VM via an interactive terminal. We can use the terminal to compile the prediction service component into a Linux container image using the *Docker* command-line tools. We can then upload the compiled image to and register it with the SEASTAR application sidekick registry from within the workspace using the SEASTAR sidekick API.

Evaluation

The aim of this evaluation is not to validate the effectiveness of the machine learning models, but to demonstrate the feasibility of the use case in SEASTAR. For the evaluation of our approach, we follow a simplified version of the experimental methodology described in [Tuncer et al., 2017b]. Since our experiments are carried out in a non-production HPC testbed environment, we do not have historic application and platform telemetry readily available. Consequently, the first step of our experimental approach is to create synthetic training data using an application setup that is purposefully designed to expose synthetic node-level anomalies. In order to simplify our experimental setup while still generating relevant telemetry, we use five different MPI-based applications from the NAS Parallel Benchmarks (NPB) [Bailey et al., 1991] suite that represent different computation and communication patterns:

- `BT` — Block Tri-diagonal solver
- `CG` — Conjugate Gradient, irregular memory access and communication
- `FT` — discrete 3D fast Fourier Transform, all-to-all communication
- `LU` — Lower-Upper Gauss-Seidel solver
- `SP` — Scalar Penta-diagonal solver

For each of the five applications, we introduce three different types of node-level anomalies that are often experienced on production HPC systems. To generate the anomalies we build on the HPC Performance Anomaly Suite (HPAS) [Ates et al., 2019], which provides a set of synthetic anomalies that reproduce common root causes of performance variations in supercomputers, including CPU contention, cache evictions, memory bandwidth interference and I/O storage server contention[6]. We have chosen three different anomaly generators from the HPAS suite which are run alongside the main NPB applications:

- *Memory leak* (`memleak`): this program simulates memory exhaustion on an HPC node which will eventually result in the operating-system terminating the application process. The implementation of this program allocates memory at a configurable rate without releasing it, simulating a memory leak.
- *CPU Interference* (`cpuif`): incorrectly terminated jobs, "rogue" system processes and concurrently running jobs (shared node tenancy) can cause CPU, cache, and I/O interference, which can impact an application's performance. The implementation of this program generates random floating point numbers and performs arithmetic operations with a configurable intensity, causing CPU and cache performance degradation for the NPB application.
- *I/O starvation* (`ioif`): faulty network connections, degraded disk arrays, and intense I/O operations of other processes can cause an application's read and write performance to degrade significantly. The implementation of this program executes configurable filesystem operations causing reduced I/O capacity of the node.

In order to test the anomaly detection service, we run of each of the five NPB applications ten times without any synthetic anomalies, ten times with (`memleak`), ten times with (`cpuif`), and ten times with (`ioif`) running in parallel. Since some NPB applications require the number of MPI ranks to be the square of an integer or to be a power of two, we configured the applications to use 64 MPI ranks, which map to 32 2-core nodes in our testbed cluster. All applications were configured as NPB problem size *Class C*, which is summarized in table 5.1. We repeat each configuration 10 times, each time with a certain amount of randomness introduced to the configuration (i.e., intensity) of the synthetic anomaly programs. This leads to a total of 2000 application runs on our testbed cluster. Telemetry for all runs is continuously collected by SEASTAR and tagged with the type of NPB application, the name of the anomaly program (or none), and the (random) configuration of the anomaly program. The telemetry collection

[6]The HPAS suite is available online `https://github.com/peaclab/hpas`

Benchmark	**Parameter**	**Problem Size (Class C)**
BT	grid size	162 x 162 x 162
	no. of iterations	200
	time step	0.0001
CG	no. of rows	150000
	no. of nonzeros	15
	no. of iterations	75
	eigenvalue shift	110
FT	grid size	512 x 512 x 512
	no. of iterations	20
LU	grid size	162 x 162 x 162
	no. of iterations	250
	time step	2.0
SP	grid size	162 x 162 x 162
	no. of iterations	400
	time step	0.00067

Table 5.1: Overview of NPB class C problem sizes and parameters used for training data generation.

agents have been configured to take samples every one second. The combined 500 hours of telemetry data with a sample frequency of 1 second resulted in 200 GB of uncompressed raw data in the telemetry data lake. Note that only a small subset of the data was used for feature selection.

After generating the training data, the next step is preprocessing. As suggested in [Tuncer et al., 2017b], we remove the first and the last 30 seconds of each telemetry time series to remove the initialization and termination phases of the applications. This duration is specific to the NBP application used in this experiment. The preprocessing is executed in parallel by loading the data from S3 directly into PySpark `DataFrames`. Next, we use PySpark ML's `DecisionTreeClassifier` and `RandomForestClassifier` to train two different models for the anomaly predictor. Both, the `DecisionTreeClassifier` and the `RandomForestClassifier` take two arrays as input: one array holding the training samples, and one array holding the class labels for the training samples. The preprocessing and training steps take approximately 5 minutes for DT and RF on an EMR cluster with 4 nodes. If we add the startup time for the ad-hoc EMR cluster, the overall execution time increases by one extra minute. Compared to the performance on a single node, the parallelized model training adds a significant speed-up and allows the processing of data sets that are too big to fit into a single node's memory. This

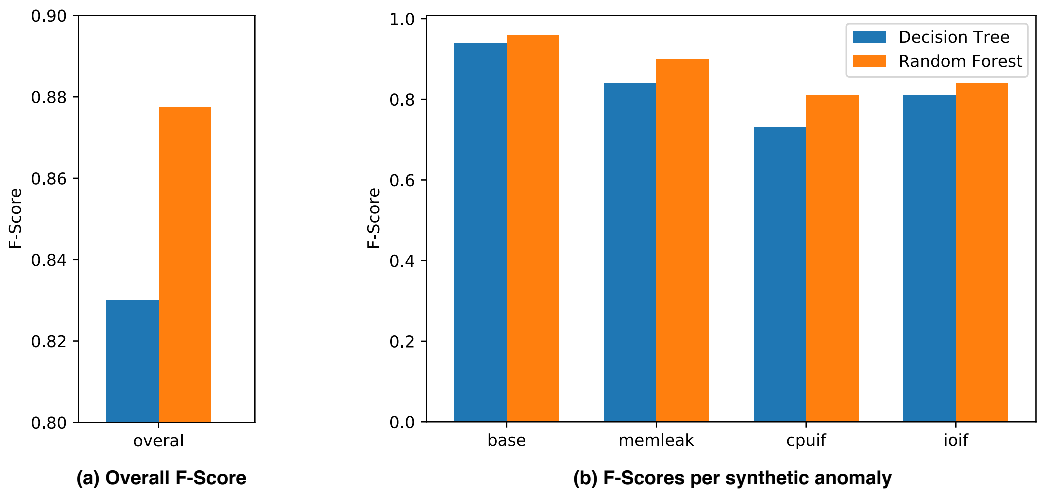


Figure 5.22: F-scores calculated for the decision tree and random forest methods for each of the four different synthetic anomaly evaluation scenarios: baseline, memory leak, CPU interference, and I/O starvation.

becomes critical if we want to scale out anomaly detection to a broader set of real-life applications and apply it to production-size HPC environments.

With the trained models in place and loaded into the anomaly prediction sidekick service, we run several scenarios, following the experimental protocol outlined in [Tuncer et al., 2017b]. To qualify the robustness of the model, we compare three different cases against a baseline case: (1) unknown anomaly patterns, (2) unknown application configuration, and (3) unknown applications. In the baseline case, we use application and anomaly program configurations that have also been used in the training data. For the unknown application configuration case, we change the configuration of the NBP application but run it with a known anomaly program configuration. Lastly, in the unknown application case, we choose an unknown NPB application and run it with a known anomaly program configuration. The results in terms of precision and recall (F-score) are summarized in figure 5.22 and are very similar to the results reported in [Tuncer et al., 2017b]. Further details on the experimental protocol and interpretation of results can be found there. [7]

We run the classification step on a 100-second data sample window. This means that we accumulate telemetry of the observed application for 100 seconds before we

[7] An interesting finding in [Tuncer et al., 2017b] is that the type of telemetry data available for model training feature selection can have a significant impact on the overall accuracy and robustness of the models. A platform like SEASTAR would provide a useful environment to explore this further.

run the Scikit-learn `predict` function on the DT and RF models. With a preconfigured telemetry ingestion frequency of 1 second, this results in 100 samples. For the given sample size, the classification step takes about 50 milliseconds on a single-threaded sidekick container node.

5.4.2 Interpretation of Results

The results obtained from the use case evaluation has shown that an existing experimental machine learning workflow can be implemented and reproduced in SEASTAR. Furthermore, we were able to show how the SEASTAR platform aids developers to turn an experimental workflow into a more robust, production-grade feedback-control system that can transparently support application users. In this sense, the evaluation presented complements existing work by adding a software architecture and engineering perspective to it. The experimental results obtained do not allow us to make any definitive claims about the overall scalability of our prototype system. However, given the well-documented scalability of Spark on AWS S3 and EMR[8], we believe that our proposed architecture and implementation can scale to large-scale HPC systems and applications. We aim to verify this in future experiments.

Using machine learning techniques for understanding an application's behaviour and building automated optimization and mitigation processes is a very promising approach for increasing application performance and for avoiding premature job terminations and wasted compute cycles. Monitoring applications and systems with established tools and methods will be stretched to their limits with the ever-growing size and complexity of HPC systems and applications. Hundreds or oven thousands of metrics collected from thousands of nodes and tens of thousands of processes at frequencies suitable for performance analysis translate to billions of data points per day, a data volume that requires increasingly sophisticated processing and analysis approaches. We believe that machine learning will play an important role in this, as it decouples the data-intensive, computationally complex and resource-intensive model generation from the online classification and prediction. With well-trained models, performance and other issues can be detected in real-time at a very low cost.

To build and evolve these systems not just in an experimental environment, a

[8]See for example [Kaplunovich and Yesha, 2018] and [Gunarathne et al., 2010] and the AWS *Big Data Blog* [Gvozdjak and Marques, 2019, Slawski and Kelly, 2019] for practical examples and performance evaluations.

telemetry platform like SEASTAR is crucial. It not only provides the data in a coherent, easily accessible, and scalable way, but it also provides the infrastructure and capabilities to support the entire exploration and development workflow, from experimentation to production deployment of services. Without these capabilities, data would have to be extracted into external data processing and analysis environments, and logic would have to be integrated either directly into HPC system software or the applications. Not only would this become increasingly impractical with the growing size and complexity of applications, but it would also significantly slow down development iterations, and hamper a wider adoption and sharing of telemetry-driven solutions. In summary, we conclude that this use case evaluation has clearly shown the value and practical importance of the telemetry platform concept.

5.5 Cost-Benefit Analysis

The use case study in the previous section provides the first set of data points and a conversation around the potential value a telemetry data platform can bring from an application *user's* and application *developer's* perspective. From an HPC platform *operator's* perspective, i.e. the perspective of the potential operator and sponsor of a telemetry platform, it is important to understand two questions to reason about the *financial* impact a telemetry platform can make on the overall operation of an HPC platform:

- **Cost**: How can we estimate the total operating cost for a telemetry platform as a function of its size and capabilities?
- **Benefit**: How can we financially quantify the value propositions of a telemetry platform.

From an operator's perspective, the benefits of operating a telemetry platform should outweigh the cost. One approach to quantify the potential benefits of a telemetry platform is to try to quantify the *avoidable costs* that is generated by inefficient HPC workload execution and can potentially be mitigated using the resiliency and optimization capabilities provided by a telemetry platform. There are two broad categories of avoidable costs associated with inefficient workload execution:

- **Compute costs** are the costs associated with inefficient HPC resource usage. If for example an application is not optimized for a specific hardware architecture, or if an application terminates unexpectedly without producing any results, the associated compute costs (often called *Service Units* or SUs) are costs that could have been avoided.

- **Labour costs** are the costs associated with the unplanned time and labour spent by application developers and users to manually analyse and mitigate application and system failure modes, and manually optimize and fine-tune applications.

Combined, these two make up for the overall avoidable costs. To what extent these costs can be avoided by providing a telemetry platform is very difficult to quantify, and we propose a more in-depth investigation into this topic as part of our future work (see section 6.3). However, based on the preliminary work we have done, it is fair to assume that a non-trivial amount of avoidable cost can be associated with lack of application resiliency and optimization, and hence are potential use cases for a telemetry platform.

We conducted a batch job exit code analysis on *Archer* [EPCC, 2019], a 118,080-core Cray XC30 system at the Edinburgh Parallel Computing Centre (EPCC) [9]. We were granted access to 12 months of batch scheduler telemetry and analysed this data for job exit codes. The results are shown in figure 5.23 and reveal that a significant fraction of jobs (24%) exit with a non-zero exit code, half of which (12%) were terminated by the scheduler due to exceeded wall-clock time. Furthermore, the data reveals that a significant fraction of service units (40%) were charged to jobs with a non-zero exit code, 36% of which were terminated by the scheduler. While this does not necessarily mean that 24% of jobs and 40% of charged service units did not yield any usable research results, it is still indicative of a potential resource wastage issue. It is not possible to say whether wall-clock time was simply underestimated or whether an unexpected behaviour of the application has caused a longer-than-expected runtime. In either case, the applications would need some sort of resiliency mechanism to cope with their abrupt termination.

If we assume an average provider cost per service unit (using one CPU core for an hour) of £0.02 [10], the total value of service units Archer can provide during 12 months is £20,701,785. If we now assume that only 10% of service units were consumed without producing any results, e.g., due to premature job termination, this would result in more than £2,000,000 worth of service units wasted during 12 months. This of

[9] `http://www.archer.ac.uk/`

[10] Based on real-world HPC financial data. Details available upon request.

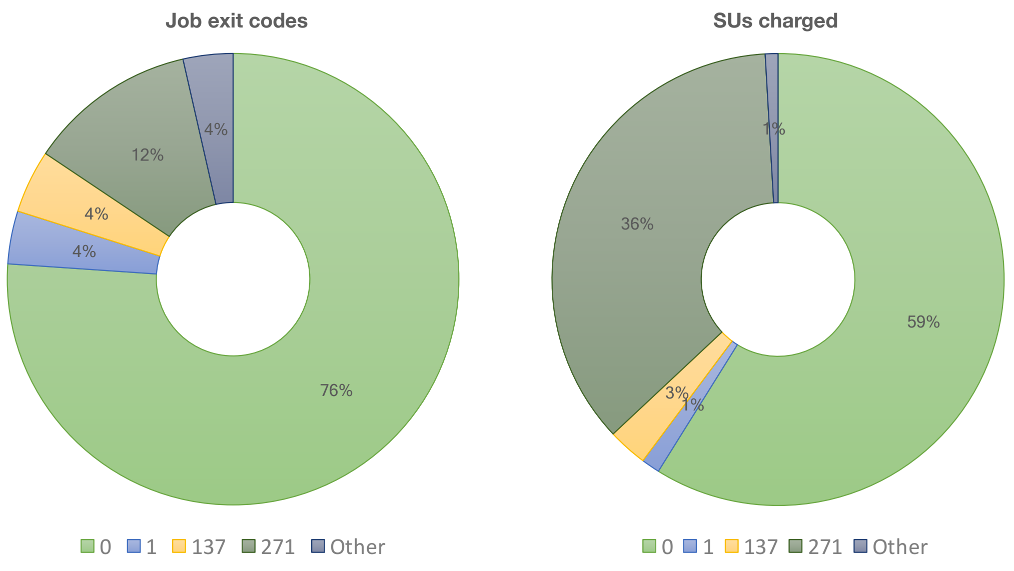


Figure 5.23: Analysis of 12 months of batch scheduler telemetry on EPCC's Archer cluster. The left chart shows the percentages of job exit codes (137 means that a job was killed, 271 means that the job was terminated due to exceeded wall-clock time limit). The chart on the right shows the percentage of service units (SUs) charged for each exit code category.

course translates to a significant amount of electrical energy wasted and greenhouse gasses emitted into the atmosphere. If we now make the conservative assumption that out of these 10% wasted service units, a fraction of 10% could be avoided by providing basic telemetry-based resiliency and optimization capabilities as proposed in this research, we end up with the approximate amount of £200,000 annually, which we could associate with the benefits of a telemetry platform deployed for Archer. Given the insights provided in figure 5.23 and the fact that we have not considered labour cost at all, the actual amount could potentially be significantly higher.

Next, we calculate a rough estimate for the cost of operating a cloud-based telemetry platform that can support a 4,920-node system like Archer. Based on the performance and scalability of AWS components outlined in the previous chapter and on the estimated data storage and transfer volumes required, we have calculated the annual cost for the operation of a telemetry data platform. As shown in table 5.2 the estimated monthly costs are around £12,800, which is equivalent to £154,248 annually. This calculation assumes AWS spot-pricing. With an up-front commitment to 12 months of resource usage, AWS grants a discount of about 30% which reduces the total an-

Service Description	**Monthly Cost**
40 TB per month S3 data lake storage.	£942
Amazon Managed Streaming for Apache Kafka (MSK) consisting of 3 m5.4xlarge Kafka broker nodes and 100 GB storage.	£4,110
Amazon EMR cluster consisting of 32 c3.2xlarge master EMR nodes, 100 %Utilized/Month.	£2,452
32 general purpose t4g.xlarge EC2 instances for TSDB, GDB, and utility service hosting.)	£2,275
AWS Fargate container platform estimated at 512 tasks or pods per day, 4 GB of memory allocated per pod, with an average job duration 2 hours	£3,075
TOTAL	**£12,854**

Table 5.2: Estimated monthly AWS cost for a telemetry platform supporting a 4,920-node system.

nual cost to about £108,000. If we compare this amount with the £200,000 we have estimated as avoidable costs, we can build a sound financial case around the return of investment of a telemetry platform. Of course, this is a very simplistic estimate, but we believe that it shows the general feasibility and direction in which this can be investigated further. As part of our future work, we would like to develop a cost function that will allow us to calculate the overall cost of a telemetry platform as a function of HPC cluster size and required telemetry platform capabilities, such as data collection frequency and storage duration, and use case categories, such as machine learning and real-time analysis. Discovering the real impact of avoidable costs needs to be investigated through more in-depth engagement with application users and engineering teams.

5.6 Summary

In this chapter we have introduced the SEASTAR platform, a prototype implementation of the conceptual telemetry platform architecture introduced in the previous chapter. Inspired by existing state-of-the-art data platform implementations, we have illustrated, how telemetry (data) platforms can be realized using a combination of open-source software components and public cloud building blocks and services. The central implementation feature of SEASTAR is its multimodal data access layer that is supported by a two-tier storage layer that covers real-time streaming, ad-hoc query, and batch processing use cases. Our description of a concrete end-to-end implementation architecture in the public AWS cloud should allow for easy reproduction and evolution of our prototype setup.

In addition to the platform architecture, we have also presented a simple programming library to make the SEASTAR data platform intuitively usable for users and developers by introducing high-level concepts and abstraction of the platform in a consistent, programmatic way. The library provides functionality for telemetry asset management, workspace management, and application sidekick management as well as access to telemetry data via the different access methods provided by the platform. One of the current shortcomings of the programming library is that it only provides access to the native interfaces of the underlying storage layer and does not provide higher-level abstractions across graph- and time-series-data as proposed by the telemetry data mode in chapter 4. We pick this up again in section 6.3 where we discuss future work. We have furthermore demonstrated, how SEASTAR can be integrated with a typical multi-tenant HPC system. For this purpose, we have built a virtual HPC testbed cluster using Amazon Web Services EC2 virtual machines and the open-source SLURM cluster manager. We have shown by example how the SLURM plug-in mechanism can be used to transparently integrate SEASTAR capabilities in existing HPC workflows and provide telemetry-based services to agnostic applications.

Lastly, we have experimentally evaluated SEASTAR's usability by implementing a machine learning-based application anomaly detection service. We were able to reproduce a previously conducted one-off experiment with our implementation and obtained very similar results. The key contribution of the use case evaluation was to complement existing work by adding a software architecture and engineering perspective to it and to show how the SEASTAR platform aids developers to turn an experimental workflow into a more robust, production-grade feedback-control system. As part of this research, we had originally implemented a second use case to illustrate SEASTAR's usability. This use case was based on the application-level-scheduling example discussed in section 2.2.2. However, since this use case illustrates the same use of components and development workflow, we have omitted it from this section.

Chapter 6

Conclusion and Future Work

Over the last ten years, data platform technology has advanced significantly. An ecosystem of complex and difficult to use technology that required expert system engineers to build and operate has emerged into a rich landscape of mature open-source projects, commercial vendors providing turnkey solutions, and cloud providers, offering data platform capabilities as a service. This development has enabled and made possible the ambitions and agenda we have set out to follow in this research. Without the commoditization of data platform technology, building a prototype for a platform like SEASTAR would not have been feasible. Our implementation impressively shows, how building a telemetry data platform becomes much more an exercise of configuring services and connecting their APIs, rather than "heavy-weight" software development. This has allowed us to focus on and explore the concepts that are important from the usability perspective of a data platform, such as workspaces and application sidekicks. The overall outcome of this work strikes in our opinion a good balance between a conceptual framework and formal model for HPC telemetry data, and a concrete implementation blueprint that sets a direction for future research and adoption of the data platform concept in practice. Despite the technological advances that have made the latter part of this work possible, presenting an end-to-end solution to the given problem space is still an enormous task. Consequently, there are many aspects in this research that have not been teased out fully and the evaluation still leaves some questions unanswered. Nevertheless, we believe that this research provides an important contribution to the field of high-performance computing platform and application architecture by demonstrating how an often overlooked, yet crucial aspect of telemetry-driven research and application design — the implementation architecture — can be supported in practice. The results we have presented, while not comprehensive, set a promising direction for future research and development of HPC telemetry platforms.

In this last chapter, we conclude our research by summarizing our key contributions, discuss the important limitations and uncertainties of our approach, and provide an outlook on several interesting and important future research topics in this area.

6.1 Results and Contributions

This research contributes to the fields of high-performance and distributed computing, particularly to the fields of HPC system and applications architecture. The four main contributions of this research are as follows:

1. **Telemetry Platform Paradigm**
   The overarching conceptual contribution of this work is the application of the data platform concept to HPC telemetry data management and usage. While existing telemetry and monitoring solutions focus on collecting telemetry data, our approach focuses on the usability of telemetry and makes analysis an integral part of the overall system instead of locating it externally. This makes the telemetry platform approach a conceptually novel approach to handling telemetry on HPC systems and distinguishes our solution from existing approaches. Furthermore, our approach caters to platform operators, application developers and users, and researchers alike. This cross-domain approach has not been taken on by existing research which tends to focus more narrowly on specific user groups or categories of use cases.

2. **Telemetry Information Model**
   The second contribution of this research is a novel telemetry data model that addresses several existing challenges of working with telemetry data by providing a time-variant structural framework, the telemetry graph, in which telemetry data can be organized and localized in a standardized way. To our knowledge, capturing the dynamic structure of and interaction between HPC platform and application components together with the telemetry data they generate has not been proposed and implemented before. It provides the larger high-performance- and distributed-computing community with a new angle on telemetry representation and a practical example of how graph-based representations of dynamic systems and applications can be used to increase the comprehensibility and comparability of system and application telemetry. The explicit distinction we make between

the abstract telemetry graph model and its concrete implementation does not only allow for the accommodation of different HPC system architectures, but it makes our approach generic enough to be applied to (distributed) systems beyond HPC systems, such as grids and clouds.

3. **Telemetry Platform Architecture**
   The third contribution of this research is the blueprint for and prototype of a concrete implementation and integration architecture of the telemetry platform paradigm and telemetry data model. Inspired by existing state-of-the-art data platform implementations, we have illustrated, how a telemetry platform can be realized using a combination of open-source software components and public cloud building blocks and services. This presents a novel approach, and we have not come across solutions that use this approach for storing and managing HPC telemetry data.

4. **Decoupled Application Architectures**
   The fourth and last contribution of this research is a proposal for decoupled HPC application architectures, separating telemetry data management, and feedback-control-loop logic from the core application code. We illustrate how this architecture pattern allows lower-complexity application code and enables the reusability of resilience and optimization capabilities that would otherwise often be tightly coupled to a specific application. We show by example how a machine learning-based, application anomaly detection service can be realized using a decoupled architecture approach. While it is safe to assume that variants of the decoupled architecture patterns are in use across many HPC application and service implementations, to our knowledge this research is the first to propose its use specifically in a telemetry-driven application context.

We believe that together, these four unique contributions provide an interesting and novel end-to-end approach to telemetry management and usage in high-performance computing, a field that has been lacking progress, compared with the speed of innovation in HPC (hardware) and application architectures. From concept to model, and from telemetry platform architecture to the decoupled architecture pattern it supports, this research motivates, touches upon and partially solves several important practical challenges in HPC telemetry: availability, accessibility, integration, standardized structure and semantics, with better support for processing and analysis.

6.2 Limitations and Uncertainties

While we think that the broad scope of this research was necessary to investigate the problem space in a meaningful way and illustrate the potential of our approach, it also resulted in a number of important details not being addressed to the extent required to build ultimate confidence in the feasibility. Especially questions about the scope in which our approach is applicable and its scalability in production HPC environments remain largely unanswered. In this section, we briefly describe these gaps. We propose a number of future work topics to address these in the subsequent section.

One important question that our research does not answer in much detail is the scope of its applicability across a broader spectrum of applications and use cases. Section 2.2 provides a comprehensive list of application areas for a telemetry platform, ranging from systems operation to adaptive application architectures, and application development. However, it does not differentiate between different types of applications and the level of granularity at which telemetry is required. For example, coarse-grained performance profiling on the task level has vastly different requirements from fine-grained communication profiling of a large MPI application. To better understand the applicability and the limits of the telemetry platform concept, future research should describe several application archetypes based on their use of telemetry, define their requirements, and match these against the platform's capabilities. We hypothesize that we will see at least two scenarios in which the telemetry platform concept will reach its limits:

- The duration between the time at which telemetry is generated and at which it becomes available on the platform is too long. This could for example be the case for extremely short and fine-grained feedback loops and generally use cases where a few seconds of latency is not acceptable.
- Four factors contribute to the telemetry data volume generated: the number of different measures collected, the sampling frequency at which they are collected, (3) the size of the system, and (4) the retention time for the telemetry data. Use-cases that require numerous metrics at a very high frequency (i.e., frequencies below 1 second) will probably not scale well. We expect that some low-level MPI profiling use cases will fall into this category.

We propose to do a more structured investigation — and experimentation — to develop a better understanding of the application archetypes for which the telemetry platform concept will work well and for which it will not. This should be part of a comprehensive analysis of the performance and scalability of our proposed architecture.

6.3 Proposed Future Work

In this section, we suggest several areas of future work that will help to further the maturity of the telemetry platform concept and to answer some of the questions that have remained unanswered in this research.

6.3.1 Cost-Benefit Analysis

Similar to the performance and scalability, the costs and benefits of a telemetry platform have not been quantified sufficiently yet. The main value proposition of a telemetry platform is that it can enable a wide variety of optimization and resilience mechanisms across the long tail of scientific applications. This in turn is expected to lead to better utilization of HPC systems and especially to a reduction of *hollow utilization*, i.e., the consumption of HPC resources without creating any usable output. On the other hand, the deployment and continuous operation of a telemetry platform would add additional costs to the HPC system operator's budget. An area of future work will be a detailed cost-benefit analysis. This can be approached in two ways:

1. *Predictive analysis*: based on the example of the EPCC study we have conducted as part of this research, a more in-depth study of *hollow utilization* can be conducted in order to quantify HPC resource waste. Combined with a model that can estimate the proliferation of optimization and resilience mechanisms over time (e.g., based on user surveys), the cost and benefit of a telemetry platform can be estimated.

2. *Real-world study*: the alternative approach to a predictive analysis would be establishing a real-world testbed in which a telemetry platform is provided as part of a production HPC system offering. Cost, adoption, and impact could be studied over an extended period in this environment.

Another aspect that would be interesting to quantify as part of a cost-benefit analysis is the impact on the software development process. Our hypothesis is that the telemetry platform concept can significantly decrease application development time by reducing the cognitive load on HPC software developers and computational scientists, and by providing reusable building blocks and services. Several approaches can be used to better understand the expected increase in efficiency, including structured interviews with developers and comparative studies. One could for example design a series of experiments in which software developers are tasked to implement different types of telemetry-based resilience and optimization patterns with and without the use of a telemetry platform.

6.3.2 Telemetry Graph Interface

One of the current gaps in the presented architecture and implementation of the telemetry platform concept is the lack of reference to and exposure of the underlying telemetry graph model. The SEASTAR prototype exposes the native query interfaces of the graph database, holding the telemetry graph, and of the time seriesdatabase, holding the time seriesdata associated with the nodes and edges of the telemetry graph. Both of these programming interfaces are completely disjoint which makes it cognitively difficult and tedious to explore structure and time seriesdata at the same time. For example, in order to get the network performance counter data from all compute nodes that run processes associated with a specific application, one first needs to query the graph database to identify the nodes:

```
MATCH  (node:CNode)-[:ALLOCATED_TO]-(proc:Process)-[:BELONGS_TO]-(app:Application)
WHERE  app.name = "MyAppIdentifier"
RETURN node, labels(node)
```

The return value of the graph query must then be used to query the time seriesdatabase for the nodes identified via a separate API:

```
SELECT derivative(sum("value"), 1s)
FROM "net.bytes_sent"
WHERE ("cnode" = IN('nodes returned from GDB query'))
```

This is neither a very intuitive way to interact with the telemetry graph programmatically, nor is it very efficient.

One interesting and relevant future area of research is to develop an abstraction layer on top of the native graph and time seriesAPIs. This abstraction layer would implement a “native” telemetry graph API that exposes the telemetry graph as a whole, i.e., both the structure and the embedded telemetry, within the same query interface. This could either be a domain-specific query language that combines both, graph query, and time seriesquery elements, or a higher-level programming framework. Such an abstraction layer would be the foundation for a more coherent and model-centric telemetry platform.

6.3.3 Decentralization and Data Locality

In its current prototype implementation, SEASTAR implements a fully centralized model for data storage and processing. That means that platform and application telemetry that is collected locally on compute nodes must be transported to and ingested into the telemetry data platform before it becomes available through the different APIs. While this works well for the use case we presented and at a moderate system and application scale, it will become inefficient in large-scale scenarios where real-time, high-frequency node-local data is required. In these types of scenarios, the overhead imposed by a centralized architecture will quickly become prohibitively expensive.

In order to alleviate this potential issue, we need to investigate how decentralization and data locality concepts can be integrated into the telemetry platform architecture. One interesting concept to explore would be edge caching of telemetry data, i.e., ensuring the *relevant* telemetry data is made readily available locally on the individual compute nodes. Ideally, this architecture would be transparent to the applications, i.e., they would use the existing programming interface to access telemetry, and the platform would manage data locality in the background and decide whether to fetch it from the central databases or a local cache. With such a decentralized system architecture in place, we can then explore advanced caching topologies and strategies, such as heuristics-based caching. A large body of research on caching exists that can be utilized and applied in this context.

6.3.4 Application to Distributed Computing

Many of the concepts of telemetry-driven application architectures are equally relevant, or even originate from the distributed computing domains like Grid and Cloud computing. Grid computing projects and initiatives like the Globus Toolkit MDS4 metadata service [Schopf et al., 2006b] and Open Grid Forum SAGA [Goodale et al., 2006] spearheaded the first generation of data models that describe HPC system properties and structures as they introduced on homogenous data models across different HPC systems. Similarly, a lot of resilient application architectures were born and popularized in distributed computing, as the complex distributed Grid and Cloud environments added another dimension of possible failures. Telemetry collection and usage in distributed computing research and application development is just as important as it is in HPC, and the challenges are very similar: solutions tend to be part of a specific application or a distributed framework and are not available as a general capability. Consequently, an interesting area of future work would be to explore the applicability of the concepts and techniques developed in this research.

6.3.5 Usability in Systems Research

One of the interesting opportunities we have pointed out in section 3.2 is HPC systems research, but the scope of our work did not allow for further investigation. As a future area of work, we propose to investigate how our telemetry platform concept can contribute to a more research friendly environment and better enable reproducible and comparable results. Part of the investigation would be a more detailed requirements analysis and design of the collaborative capabilities of the platform, such as shared workspaces and the telemetry catalogue. A specific focus could be on the publication of digital assets which we have just briefly touched upon in section 5.2.2.

6.3.6 Extension to Log-File Data

Many HPC application and platform anomaly detection systems use log-files instead of telemetry data as input for their analyses and predictions (see e.g., [Fronza et al., 2013], [Gainaru et al., 2012], and [Heien et al., 2011]). Log-file based resilience and optimization architectures are other important approaches that could be relevant to support by a system like SEASTAR. Log files are by nature more unstructured than telemetry and require different processing approaches and technologies. However, we believe that our telemetry graph could provide a useful semantic framework for organizing log-files within the structural context in which they occur. We propose to investigate this topic further to better understand if and how the telemetry platform paradigm can be extended to log-file data. It will be interesting to understand if telemetry and log data can efficiently coexist within the same conceptual framework, and the potential benefits this can bring to HPC application development.

6.4 Conclusion and Reflections

The original idea of embarking on this research was motivated by the observations and experience collected throughout a decade of work as an HPC application and framework developer. Working with HPC hardware and software was always interesting and rewarding due to the massive scale and the sophisticated concepts and technologies that make it possible to use systems of this scale efficiently. However, one of the consistent negative experiences across all the different HPC research and development projects[1] was indeed about managing platform and application telemetry data. Whether data needed to be collected and processed for debugging or optimization purposes, to evaluate and document experiments, or to build feedback-control loops (e.g., for application-level scheduling in RADICAL-Pilot [Merzky et al., 2015a]) — the process was always time-consuming, clunky, and error-prone. Platform facilities to extract and collect telemetry, if existing or accessible at all, were limited to basic system monitoring interfaces and tools and highly aggregated historic datasets that were usually not fit for the task at hand. The consequence was that a growing set of "home-grown" tooling had to be developed and customized for every new platform, to be the basis for telemetry extraction and management. And while this was a pragmatic way forward, the overhead was high, reusability low, and the experience was unsatisfactory from a software developer's perspective, especially in an environment that otherwise epitomizes the technological avant-garde in hardware and software research. Identifying these problems in [Weidner et al., 2016b], the resulting discussions with reviewers and peers, and a subsequent feature in *HPC Wire* [HPCWire, 2017], provided enough encouragement for an in-depth investigation into *"how can we do this better?"*. The HPC telemetry platform presented in this research is our answer to that question.

Setting out to answer this question was ambitious and the result, as presented in this thesis, shows the consequences of the broad scope that was necessary to investigate the problem space holistically: some important details were not discussed at the depth required and the experimental evaluation of the system is not comprehensive enough to build strong confidence in the feasibility of our approach. However, it paints in broad but concise strokes the picture of a new paradigm for integrating telemetry with HPC application architectures, development processes and research workflows.

[1]Published HPC application and framework development projects included [Hossain et al., 2019], [Merzky et al., 2015c], [Merzky et al., 2015a], [Balasubramanian et al., 2016], [Radak et al., 2013b], [Jha et al., 2007b], [Jha et al., 2007c], and [Weidner and Bidal, 2008].

The SEASTAR prototype, while put together only roughly for evaluation, provides a practical blueprint for implementing and integrating this paradigm in a real-world scenario, using readily available cloud computing capabilities. Expanding on or rebuilding SEASTAR should be a straight-forward exercise starting from the details developed in this research.

Some aspects of the telemetry model also did not receive the focus that they deserve, and readers might ask themselves *"why?"* after working through its formal definition, only to end up with a superficial practical example. The key idea, which unfortunately had to fall short in the prototype and its practical evaluation, is the semantic skeleton the telemetry graph provides for telemetry. Not only does this allow for organizing telemetry in its structural context, which makes it easily identifiable and comparable, but it also provides a navigable *digital twin* of a platform and the applications running on it. This is a critical capability for self-adapting applications that require understanding of the architecture and configuration of the platform they run on, as well as their locality within that structure. Many existing HPC applications and frameworks still maintain static configuration maps for known systems or require this information to be passed as parameters at startup in order to build an internal representation of their context. The telemetry graph makes this context discoverable for applications at runtime, which provides great opportunities for novel adaptive application architectures, which could have easily been turned into an additional chapter. We hope to elaborate this further in an upcoming journal publication.

But while the list of open topics is long, we still believe strongly that this research makes an important novel contribution towards furthering our understanding of future HPC ecosystem designs and architectures. We hope that this research will inspire future research and development of HPC telemetry platforms. We are confident that, if this research concept can evolve into a commodity HPC system capability, it has the potential to improve research transparency and collaboration, ease proliferation of advanced, telemetry-driven application architectures, and ultimately platform efficiency, beyond what is currently feasible.

Appendix A

Telemetry Usage Survey

From the existing literature, it is difficult to distil a comprehensive picture of how telemetry data is used across production HPC systems. In the majority of cases, experimental workflow descriptions simply assume that telemetry is available. How it is collected and accessed remains vague at best. We set out to do a survey that explores how telemetry data is managed and used at HPC centres and how it is made available to the users[1]. Unfortunately, the response rate to our survey was very low. Only three out of 20 selected HPC centres responded to our request, this is why we have decided not to give it a more prominent place in our research. We suspect that the fact that we have sent the survey to publicly available HPC centre help-desk email addresses instead of individuals has contributed to the low number of participants. We are planning for a second iteration with an updated set of questions that will be sent out to individuals instead of mailing lists as part of our future work.

A.1 Survey Design

The survey consists of a catalogue of ten questions:

1. Which open-source monitoring tools are integrated with your platform? (Please select all that apply)
2. Which vendor-specific monitoring tools are integrated with your platform? (Please select all that apply)
3. What kind of data do you collect? (Please select all that apply)

[1]Survey accessible online at `https://www.surveymonkey.com/r/RHRQXNW`

4. What do you use telemetry data for? (Please select all that apply)
5. Do you provide programmatic (API) access to telemetry data for application developers?
6. If you answered the previous question with ”Yes”, can you describe the API you provide?
7. Are you aware of any applications on your platform that use the telemetry data you collect/provide?
8. If you answered the previous question with ”Yes”, can you give examples of which applications use the data and how?
9. Are you aware of any applications on your platform that collect and use telemetry on their own?
10. Can you share the name of the HPC system for which the answers above apply (optional)?

A.2 Results

The survey was sent out to 20 HPC centres around the globe. The selection was guided by the November 2017 TOP500 list [2]. The total number of responses was three. While none-representative, it was still interesting that all three participants answered question 5 — *Do you provide programmatic (API) access to telemetry data for application developers* with “No”. This is consistent with our own experience with the HPC platforms we have worked on. The full results are shown below.

[2]`https://www.top500.org/lists/2017/11/`

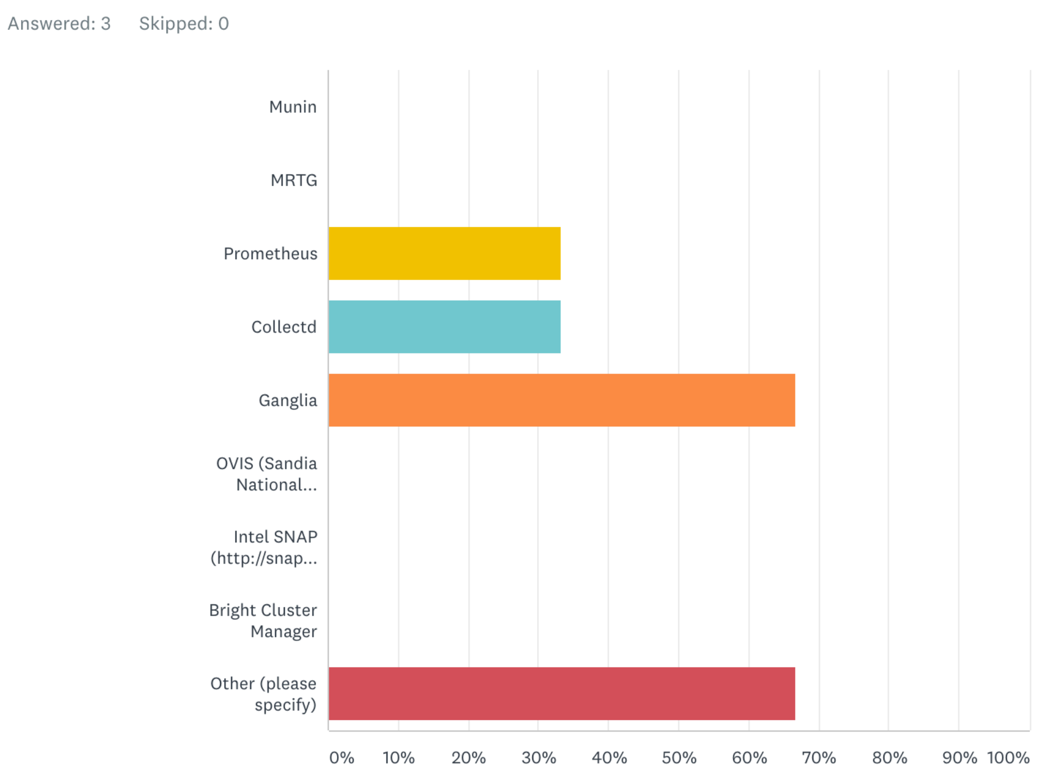


Figure A.1: Question 1: Which open-source monitoring tools are integrated with your platform? (Please select all that apply)

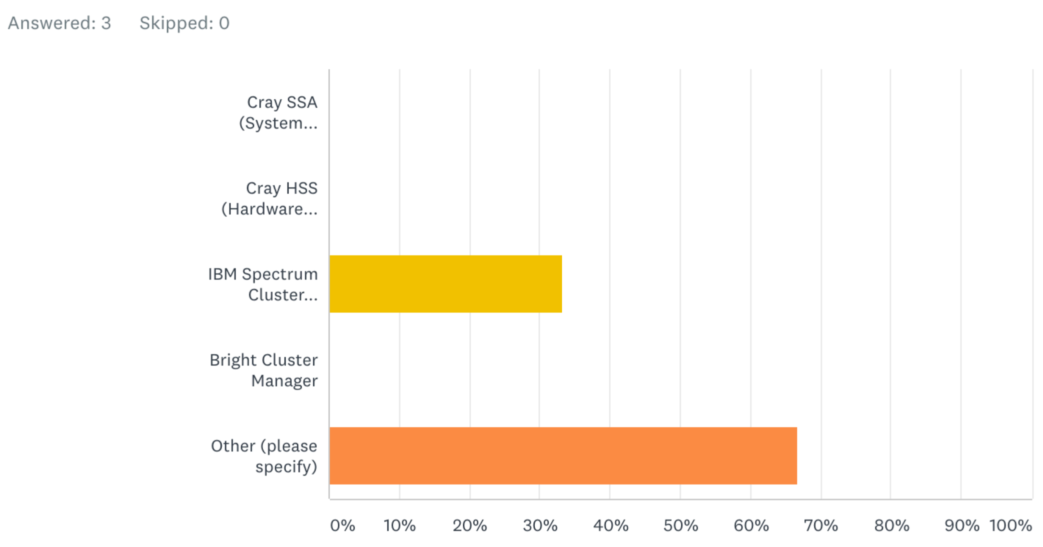


Figure A.2: Question 2: Which vendor-specific monitoring tools are integrated with your platform? (Please select all that apply)

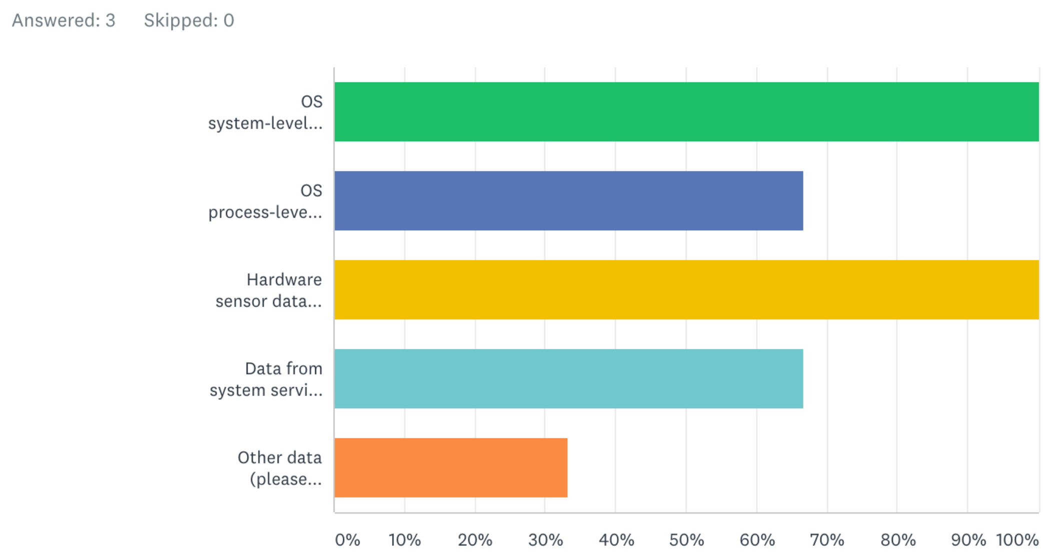


Figure A.3: Question 3: What kind of data do you collect? (Please select all that apply)

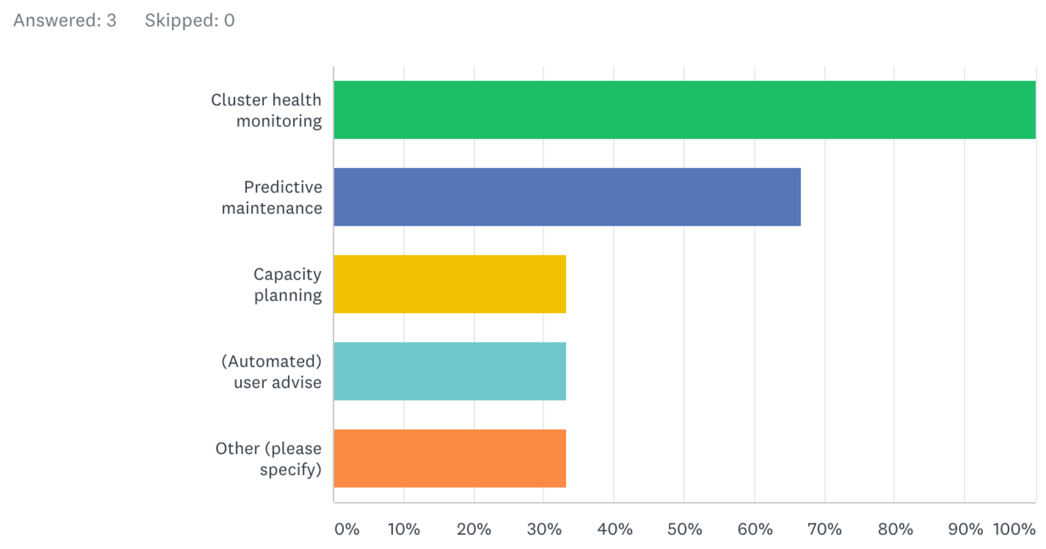


Figure A.4: Question 4: What do you use telemetry data for? (Please select all that apply)

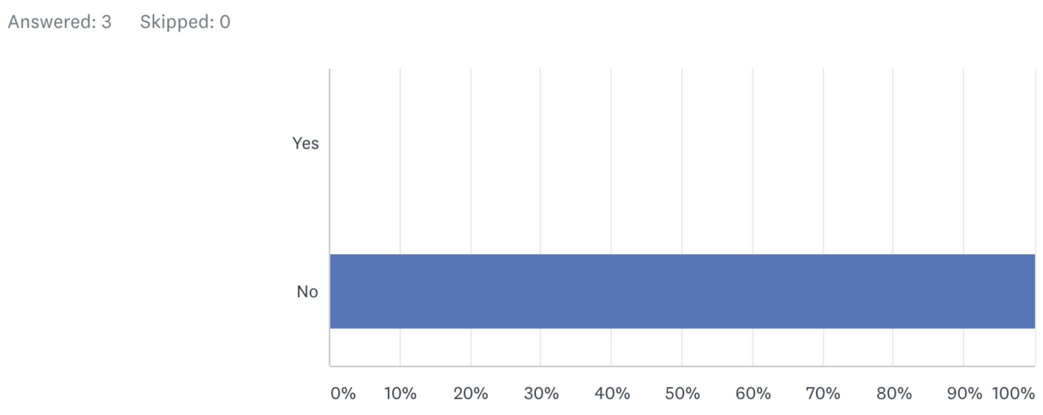


Figure A.5: Question 5: Do you provide programmatic (API) access to telemetry data for application developers?

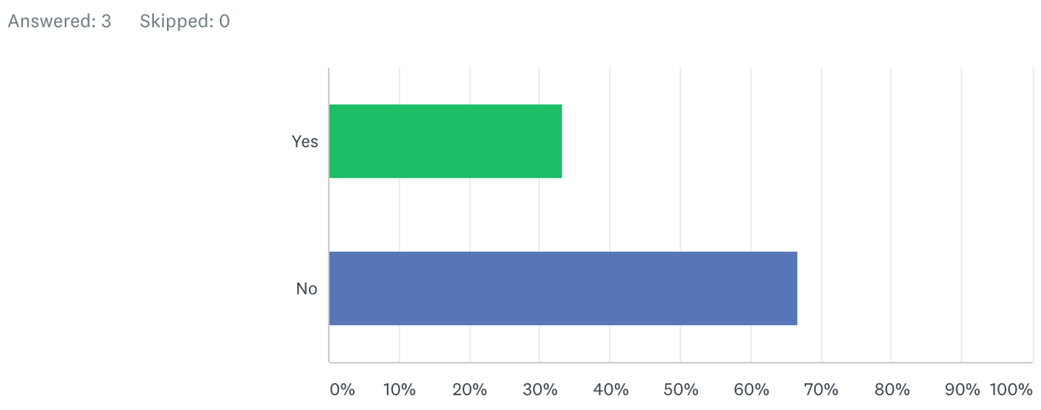


Figure A.6: Question 7: Are you aware of any applications on your platform that use the telemetry data you collect / provide?

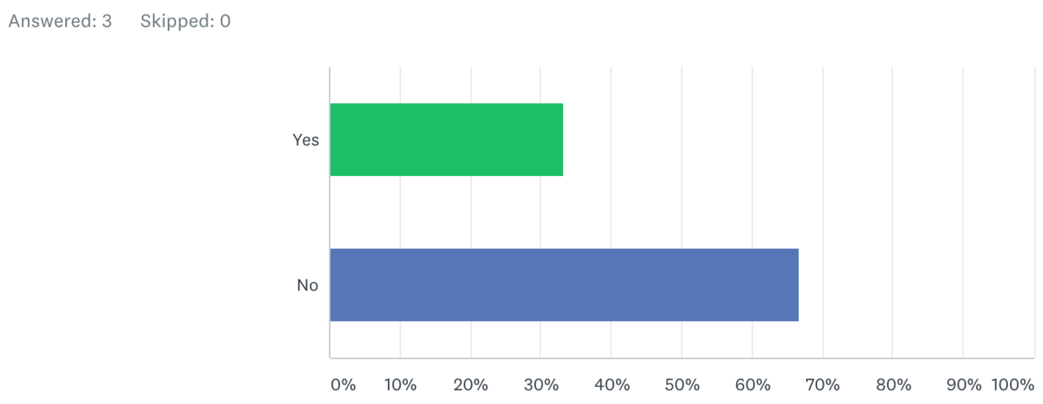


Figure A.7: Question 9: Are you aware of any applications on your platform that collect and use telemetry on their own?

Glossary

application anatomy

Application anatomy describes a type of subgraph of a telemetry graph that represents structure and properties of HPC applications. 75

application sidekick

Application sidekicks are a component of a telemetry platform that provides capabilities for building and hosting telemetry-based application support services. 117

application telemetry

Application telemetry describes a subset of telemetry that is generated during the execution of applications on an HPC cluster. This includes information about an application's system resource allocation and interaction as well as information generated by the application itself, such as internal performance metrics. 18

data lake

A data lake is a system that provides a repository of data stored in its natural/raw format, usually object blobs or files. 5

platform anatomy

Platform anatomy describes a type of subgraph of a telemetry graph that represents structure and properties of the HPC system. 75

system telemetry

System telemetry describes a subset of telemetry that is generated by the operating systems, commanding compute, storage, networking and utility nodes, environmental sensors that monitor power consumption, temperature, and other external factors, and HPC system services such as job queueing and object storage systems. 18

telemetry

Telemetry is the continuous stream of operational data that is generated on HPC systems by the hardware, operating systems, services, runtime systems, and applications 1

telemetry graph

A telemetry graph is a labelled, directed multigraph that represents the structure of HPC systems and applications. It provides the semantic structure in which telemetry is organized. 76

Acronyms

AMR

Adaptive Mesh Refinement 29

API

Application programming interface 3

AWS

Amazon Web Service 11

DPaaS

Data-Platform-as-a-Service 52

ETL

Extract, Transform, Load 103

FPGA

Field-Programmable Gate Array 36

GDB

Graph Database 90

GPU

Graphics Processing Unit 36

HPC

High-Performance Computing 1

IoT

internet-of-Things 2

JVM

JAVA Virtual Machine 28

MPI

Message Passing Interface 17

PaaS

Platform-as-a-Service 52

ProcFS

Process Filesystem 19

RDBMS

Relational Database Management System 90

REST

Representational state transfer 60

TSDB

Time-Series Database 90

XDR

External Data Representation 33

XML

Extensible Markup Language 33

Bibliography

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.

[Agelastos et al., 2014] Agelastos, A., Allan, B., Brandt, J., Cassella, P., Enos, J., Fullop, J., Gentile, A., Monk, S., Naksinehaboon, N., Ogden, J., Rajan, M., Showerman, M., Stevenson, J., Taerat, N., and Tucker, T. (2014). The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165.

[Agrawal et al., 2014] Agrawal, K., Fahey, M. R., McLay, R. T., and James, D. (2014). User environment tracking and problem detection with XALT.

[Ahmed et al., 2016] Ahmed, M., Mahmood, A. N., and Islam, M. R. (2016). A survey of anomaly detection techniques in financial domain. *Future Generation Computer Systems*, 55:278–288.

[Allaire, 2012] Allaire, J. (2012). Rstudio: integrated development environment for r. *Boston, MA*, 770:394.

[Allcock et al., 2011] Allcock, W., Felix, E., Analysis, M. L. S., , and 2011 (2011). Challenges of HPC monitoring. *ieeexplore.ieee.org*.

[Amundsen, 2020] Amundsen (2020). Amundsen, open source data discovery and metadata engine. `https://www.amundsen.io/`. [Online; accessed 16-Dec-2020].

[Ates et al., 2019] Ates, E., Zhang, Y., Aksar, B., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2019). HPAS: An HPC performance anomaly suite for reproducing performance variations. In *48th International Conference on Parallel Processing (ICPP 2019)*.

[Atkinson et al., 2017] Atkinson, M., Gesing, S., Montagnat, J., and Taylor, I. (2017). Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216–227.

[Bader et al., 2017] Bader, A., Kopp, O., and Falkenthal, M. (2017). Survey and comparison of open-source time seriesdatabases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband.*

[Bailey et al., 1991] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

[Balasubramanian et al., 2016] Balasubramanian, V., Treikalis, A., Weidner, O., and Jha, S. (2016). Ensemble toolkit: Scalable and flexible execution of ensembles of tasks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 458–463. IEEE.

[Baltrušaitis et al., 2018] Baltrušaitis, T., Ahuja, C., and Morency, L.-P. (2018). Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443.

[Barroso and Hölzle, 2007] Barroso, L. A. and Hölzle, U. (2007). The case for energy-proportional computing. *Computer*, 40(12):33–37.

[Bechhofer et al., 2010] Bechhofer, S., De Roure, D., Gamble, M., Goble, C., and Buchan, I. (2010). Research objects: Towards exchange and reuse of digital knowledge. *Nature Precedings*, pages 1–1.

[Becker et al., 1995] Becker, D. J., Sterling, T., Savarese, D., Dorband, J. E., Ranawak, U. A., and Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, volume 95, pages 11–14.

[Beckman et al., 2007] Beckman, P., Nadella, S., Trebon, N., and Beschastnikh, I. (2007). Spruce: A system for supporting urgent high-performance computing. In Gaffney, P. W. and Pool, J. C. T., editors, *Grid-Based Problem Solving Environments*, pages 295–311, Boston, MA. Springer US.

[Bernstein, 2014] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[Berral et al., 2010] Berral, J. L., Goiri, I. n., Nou, R., Julià, F., Guitart, J., Gavaldà, R., and Torres, J. (2010). Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, page 215–224, New York, NY, USA. Association for Computing Machinery.

[Bhatele et al., 2015] Bhatele, A., Titus, A. R., Thiagarajan, J. J., Jain, N., Gamblin, T., Bremer, P., Schulz, M., and Kale, L. V. (2015). Identifying the culprits behind network congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 113–122.

[Bhuyan et al., 2013] Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. (2013). Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials*, 16(1):303–336.

[Bianchini and Rajamony, 2004] Bianchini, R. and Rajamony, R. (2004). Power and energy management for server systems. *Computer*, 37(11):68–76.

[Bollobás, 2013] Bollobás, B. (2013). *Modern graph theory*, volume 184. Springer Science & Business Media.

[Borghesi et al., 2019] Borghesi, A., Libri, A., Benini, L., and Bartolini, A. (2019). Online anomaly detection in hpc systems. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 229–233. IEEE.

[Browne et al., 2000] Browne, S., Dongarra, J., Garner, N., London, K., and Mucci, P. (2000). A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 42–42.

[CACM, 2017] CACM (2017). Rethinking HPC systems for 'Second Gen' Applications. `https://cacm.acm.org/news/213986-rethinking-hpc-platforms-for-second-gen-applications/fulltext`. [Online; accessed 16-Jan-2018].

[Chandola et al., 2009] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58.

[Chatfield, 1978] Chatfield, C. (1978). The holt-winters forecasting procedure. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 27(3):264–279.

[Chung-hsing Hsu and Wu-chun Feng, 2005] Chung-hsing Hsu and Wu-chun Feng (2005). A power-aware run-time system for high-performance computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 1–1.

[CKAN, 2020] CKAN (2020). CKAN, the world's leading Open Source data portal platform. `https://ckan.org/`. [Online; accessed 16-Dec-2020].

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

[Datadog, 2021] Datadog (2021). Datadog, cloud monitoring as a service. `https://www.datadoghq.com/`. [Online; accessed 05-Sept-2021].

[Davison, 2003] Davison, A. C. (2003). *Statistical models*, volume 11. Cambridge university press.

[Dean and Ghemawat, 2010] Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77.

[Deelman et al., 2015] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., Da Silva, R. F., Livny, M., et al. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.

[Diestel, 2000] Diestel, R. (2000). *Graduate Texts in Mathematics, Volume 173*. Springer-Verlag New York, Incorporated.

[Dorier et al., 2014] Dorier, M., Antoniu, G., Ross, R., Kimpe, D., and Ibrahim, S. (2014). Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164.

[Duckworth et al., 2017] Duckworth, J., Blakeborough, J., Coryell, K., and McLeod, S. (2017). Telemetry-enabled customer support using the cray system snapshot analyzer (ssa). `https://cug.org/proceedings/cug2017_proceedings/includes/files/pap124s2-file1.pdf`. [Online; accessed 10-Apr-2020].

[EEHPCWG, 2014] EEHPCWG (2014). Energy Efficient High Performance Computing Working Group. `https://eehpcwg.llnl.gov//`. [Online; accessed 10-Apr-2020].

[Ellert et al., 2007] Ellert, M., Grønager, M., Konstantinov, A., Kónya, B., Lindemann, J., Livenson, I., Nielsen, J. L., Niinimäki, M., Smirnova, O., and Wäänänen, A. (2007). Advanced resource connector middleware for lightweight computational grids. *Future Generation computer systems*, 23(2):219–240.

[EPCC, 2019] EPCC (2019). Archer. `http://www.archer.ac.uk/about-archer/`. [Online; accessed 03-Apr-2020].

[Faulkner and Gomes, 1991] Faulkner, R. and Gomes, R. (1991). The process file system and process model in unix system v. In *USENIX Winter*.

[Fernandes and Bernardino, 2018] Fernandes, D. and Bernardino, J. (2018). Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb. In *Data*, pages 373–380.

[Ferreira, 2021] Ferreira, R. (2021). Take me down to the paradise city where the metric is green and traces are pretty. USENIX Association.

[Filgueira et al., 2010] Filgueira, R., Singh, D. E., Carretero, J., Calderón, A., and García, F. (2010). Adaptive-compi: Enhancing mpi-based applicationsâ performance and scalability by using adaptive compression. *International Journal of High Performance Computing Applications*.

[Filgueira et al., 2011] Filgueira, R., Singh, D. E., Carretero, J., Calderón, A., and García, F. (2011). Adaptive-Compi: Enhancing Mpi-Based Applications' Performance and Scalability by using Adaptive Compression. *The International Journal of High Performance Computing Applications*, 25(1):93–114.

[Fontenla-Romero et al., 2013] Fontenla-Romero, Ó., Guijarro-Berdiñas, B., Martinez-Rego, D., Pérez-Sánchez, B., and Peteiro-Barral, D. (2013). Online machine learning. In *Efficiency and Scalability Methods for Computational Intellect*, pages 27–54. IGI Global.

[Foster, 2006] Foster, I. (2006). Globus toolkit version 4: Software for service-oriented systems. *Journal of computer science and technology*, 21(4):513–520.

[Foundation, 2021] Foundation, T. L. (2021). Open Neural Network Exchange: The open standard for machine learning interoperability. `https://onnx.ai/`. [Online; accessed 021-Jan-2021].

[Francis et al., 2018] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445.

[Fronza et al., 2013] Fronza, I., Sillitti, A., Succi, G., Terho, M., and Vlasenko, J. (2013). Failure prediction based on log files using random indexing and support vector machines. *Journal of Systems and Software*, 86(1):2–11.

[Fürlinger and Gerndt, 2005] Fürlinger, K. and Gerndt, M. (2005). ompp: A profiling tool for openmp. In *International Workshop on OpenMP*, pages 15–23. Springer.

[Gainaru et al., 2012] Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2012). Fault prediction under the microscope: A closer look into hpc systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE.

[Gallo et al., 2015] Gallo, S. M., White, J. P., DeLeon, R. L., Furlani, T. R., Ngo, H., Patra, A. K., Jones, M. D., Palmer, J. T., Simakov, N., Sperhac, J. M., Innus, M., Yearke, T., and Rathsam, R. (2015). Analysis of xdmod/supremm data using machine learning techniques. In *2015 IEEE International Conference on Cluster Computing*, pages 642–649.

[Gandrud, 2013] Gandrud, C. (2013). *Reproducible research with R and R studio*. CRC Press.

[Ge et al., 2005] Ge, R., Xizhou Feng, and Cameron, K. W. (2005). Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 34–34.

[Goodale et al., 2006] Goodale, T., Jha, S., Kaiser, H., Kielmann, T., Kleijer, P., Von Laszewski, G., Lee, C., Merzky, A., Rajic, H., and Shalf, J. (2006). Saga: A simple api for grid applications. high-level application programming on the grid. *Computational Methods in Science and Technology*, 12(1):7–20.

[GrafanaLabs, 2020] GrafanaLabs (2020). Grafana. `https://grafana.com/`. [Online; accessed 16-Jan-2018].

[Granger and Grout, 2016] Granger, B. and Grout, J. (2016). Jupyterlab: Building blocks for interactive computing. *Slides of presentation made at SciPy*.

[Gropp et al., 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

[Gunarathne et al., 2010] Gunarathne, T., Wu, T., Qiu, J., and Fox, G. (2010). Mapreduce in the clouds for science. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 565–572.

[Gvozdjak and Marques, 2019] Gvozdjak, P. and Marques, J. (2019). Amazon EMR introduces EMR runtime for Apache Spark. `https://aws.amazon.com/blogs/big-data/amazon-emr-introduces-emr-runtime-for-apache-spark/`. [Online; accessed 16-Jan-2021].

[Hargrove and Duell, 2006] Hargrove, P. H. and Duell, J. C. (2006). Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494.

[Heien et al., 2011] Heien, E., Kondo, D., Gainaru, A., LaPine, D., Kramer, B., and Cappello, F. (2011). Modeling and tolerating heterogeneous failures in large parallel systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA. Association for Computing Machinery.

[Herbein et al., 2016] Herbein, S., Ahn, D. H., Lipari, D., Scogland, T. R., Stearman, M., Grondona, M., Garlick, J., Springmeyer, B., and Taufer, M. (2016). Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 69–80, New York, NY, USA. Association for Computing Machinery.

[Hey et al., 2009] Hey, T., Tansley, S., Tolle, K., et al. (2009). *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA.

[Holzschuher and Peinl, 2013] Holzschuher, F. and Peinl, R. (2013). Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, page 195–204, New York, NY, USA. Association for Computing Machinery.

[Hossain et al., 2019] Hossain, A., Jha, S., and Weidner, O. (2019). Federation and interoperability use cases, version 1.1. Technical report.

[HPCWire, 2017] HPCWire (2017). Rethinking HPC systems for 'Second Gen' Applications. `https://www.hpcwire.com/2017/02/22/rethinking-hpc-platforms-second-gen-applications/`. [Online; accessed 16-Jan-2018].

[Hunold, 2015] Hunold, S. (2015). A Survey on Reproducibility in Parallel Computing. *CoRR*.

[Ibidunmoye et al., 2015] Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. (2015). Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1).

[InfluxData, 2020a] InfluxData (2020a). Procstat input plugin. `https://github.com/influxdata/telegraf/tree/release-1.14/plugins/inputs/procstat`.

[InfluxData, 2020b] InfluxData (2020b). Telegraf. `https://github.com/influxdata/telegraf/tree/release-1.14/`.

[Ivie and Thain, 2018] Ivie, P. and Thain, D. (2018). Reproducibility in scientific computing. *ACM Comput. Surv.*, 51(3).

[Jha et al., 2007a] Jha, S., Kaiser, H., El-Khamra, Y., and Weidner, O. (2007a). Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus. *eScience*.

[Jha et al., 2007b] Jha, S., Kaiser, H., Khamra, Y. E., and Weidner, O. (2007b). Design and implementation of network performance aware applications using saga and cactus. In *e-Science and Grid Computing, IEEE International Conference on*, pages 143–150. IEEE.

[Jha et al., 2007c] Jha, S., Kaiser, H., Merzky, A., and Weidner, O. (2007c). Grid interoperability at the application level using saga. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 584–591. IEEE.

[Josephsen, 2007] Josephsen, D. (2007). *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, USA.

[Juve et al., 2015] Juve, G., Tovar, B., d. Silva, R. F., Król, D., Thain, D., Deelman, E., Allcock, W., and Livny, M. (2015). Practical resource monitoring for robust high throughput computing. In *2015 IEEE International Conference on Cluster Computing*, pages 650–657.

[Kale and Krishnan, 1993] Kale, L. V. and Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108.

[Kaplan et al., 2013] Kaplan, F., Meng, J., and Coskun, A. K. (2013). Optimizing communication and cooling costs in hpc data centers via intelligent job allocation. In *2013 International Green Computing Conference Proceedings*, pages 1–10.

[Kaplunovich and Yesha, 2018] Kaplunovich, A. and Yesha, Y. (2018). Consolidating billions of taxi rides with aws emr and spark in the cloud : Tuning, analytics and best practices. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4501–4507.

[Kasick et al., 2010] Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P. (2010). Black-box problem diagnosis in parallel file systems. In *FAST*, pages 43–56.

[Kiran et al., 2015] Kiran, M., Murphy, P., Monga, I., Dugan, J., and Baveja, S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792.

[Kocher, 1996] Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer.

[Kónya and Johansson, 2010] Kónya, B. and Johansson, D. (2010). The nordugrid/arc information system. *The NorduGrid Collaboration. URL http://www. nordugrid. org/documents/arc_infosys. pdf. NORDUGRID-TECH-4*.

[Kreps et al., 2011] Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7.

[Liang et al., 2020] Liang, L., Filguiera, R., and Yan, Y. (2020). Adaptive optimizations for stream-based workflows. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 33–40. IEEE.

[Lovrić et al., 2019] Lovrić, M., Molero, J. M., and Kern, R. (2019). Pyspark and rdkit: Moving towards big data in cheminformatics. *Molecular Informatics*, 38(6):1800082.

[Lysoněk, 2019] Lysoněk, O. (2019). Lm_sensors - Linux hardware monitoring. `https://hwmon.wiki.kernel.org/lm_sensors`. [Online; accessed 03-Apr-2020].

[Massie et al., 2004] Massie, M. L., Chun, B. N., and Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840.

[Meng et al., 2015] Meng, J., McCauley, S., Kaplan, F., Leung, V. J., and Coskun, A. K. (2015). Simulation and optimization of hpc job allocation for jointly reducing communication and cooling costs. *Sustainable Computing: Informatics and Systems*, 6:48–57. Special Issue on Selected Papers from 2013 International Green Computing Conference (IGCC).

[Meng et al., 2016] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.

[Merzky et al., 2015a] Merzky, A., Santcroos, M., Turilli, M., and Jha, S. (2015a). Executing Dynamic and Heterogeneous Workloads on Super Computers. *arXiv.org*.

[Merzky et al., 2015b] Merzky, A., Santcroos, M., Turilli, M., and Jha, S. (2015b). Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers. *CoRR, abs/1512.08194*.

[Merzky et al., 2015c] Merzky, A., Weidner, O., and Jha, S. (2015c). Saga: a standardized access layer to heterogeneous distributed computing infrastructure. *SoftwareX*, 1:3–8.

[Miller et al., 1995] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. (1995). The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46.

[Miloslavskaya and Tolstoy, 2016] Miloslavskaya, N. and Tolstoy, A. (2016). Big data, fast data and data lake concepts. *Procedia Computer Science*, 88:300–305.

[Mohr and Wolf, 2003] Mohr, B. and Wolf, F. (2003). Kojak – a tool set for automatic performance analysis of parallel programs. In Kosch, H., Böszörményi, L., and Hellwagner, H., editors, *Euro-Par 2003 Parallel Processing*, pages 1301–1304, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Moore et al., 2015] Moore, C. L., Khalsa, P. S., Yilk, T. A., and Mason, M. (2015). Monitoring high performance computing systems for the end user. In *2015 IEEE International Conference on Cluster Computing*, pages 714–716.

[Myagmar et al., 2005] Myagmar, S., Lee, A. J., and Yurcik, W. (2005). Threat modeling as a basis for security requirements. In *Symposium on requirements engineering for information security (SREIS)*, volume 2005, pages 1–8. Citeseer.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads programming*. O'Reilly & Associates, Inc.

[Oetiker, 2017] Oetiker, T. (2017). RRDTool. `https://oss.oetiker.ch/rrdtool/`. [Online; accessed 16-Jan-2018].

[Palmer et al., 2015] Palmer, J. T., Gallo, S. M., Furlani, T. R., Jones, M. D., DeLeon, R. L., White, J. P., Simakov, N., Patra, A. K., Sperhac, J., Yearke, T., Rathsam, R., Innus, M., Cornelius, C. D., Browne, J. C., Barth, W. L., and Evans, R. T. (2015). Open xdmod: A tool for the comprehensive management of high-performance computing resources. *Computing in Science Engineering*, 17(4):52–62.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Powers, 2020] Powers, D. (2020). Kafka python client. `https://github.com/dpkp/kafka-python`.

[Radak et al., 2013a] Radak, B. K., Lee, T.-S., He, P., Romanus, M., Weidner, O., Dai, W., Gallicchio, E., Deng, N.-J., York, D. M., Levy, R. M., and Jha, S. (2013a). A framework for flexible and scalable replica-exchange on production distributed CI. *XSEDE*, page 1.

[Radak et al., 2013b] Radak, B. K., Romanus, M., Gallicchio, E., Lee, T.-S., Weidner, O., Deng, N.-J., He, P., Dai, W., York, D. M., Levy, R. M., et al. (2013b). A framework for flexible and scalable replica-exchange on production distributed ci. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 26. ACM.

[Raj, 2015] Raj, S. (2015). *Neo4j high-performance*. Packt Publishing Ltd.

[Ritchie and Thompson, 1978] Ritchie, D. M. and Thompson, K. (1978). The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929.

[Schopf et al., 2006a] Schopf, J. M., Pearlman, L., Miller, N., Kesselman, C., Foster, I., D'Arcy, M., and Chervenak, A. (2006a). Monitoring the grid with the globus toolkit mds4. In *Journal of Physics: Conference Series*, volume 46, page 072. IOP Publishing.

[Schopf et al., 2006b] Schopf, J. M., Pearlman, L., Miller, N., Kesselman, C., Foster, I., D'Arcy, M., and Chervenak, A. (2006b). Monitoring the grid with the globus toolkit mds4. *Journal of Physics: Conference Series*, 46(1):521.

[Services, 2021] Services, A. W. (2021). Amazon cloudwatch, application and infrastructure monitoring. `https://aws.amazon.com/cloudwatch/`. [Online; accessed 05-Sept-2021].

[Sethi et al., 2019] Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., and Berner, C. (2019). Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813.

[Shvachko et al., 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10.

[Slawski and Kelly, 2019] Slawski, P. and Kelly, J. (2019). Improve Apache Spark write performance on Apache Parquet formats with the EMRFS S3-optimized committer. `https://aws.amazon.com/blogs/big-data/improve-apache-spark-write-performance-on-apache-parquet-formats-with-the-emrfs-s3-optimized-committer/`. [Online; accessed 16-Jan-2021].

[Sottile and Minnich, 2002] Sottile, M. J. and Minnich, R. G. (2002). Supermon: a high-speed cluster monitoring system. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 39–46.

[Stubbs et al., 2020] Stubbs, J., Looney, J., Poindexter, M., Chalhoub, E., Zynda, G. J., Ferlanti, E. S., Vaughn, M., Fonner, J. M., and Dahan, M. (2020). Integrating jupyter into research computing ecosystems: Challenges and successes in architecting jupyterhub for collaborative research computing ecosystems. In *Practice and Experience in Advanced Research Computing*, pages 91–98.

[Thain et al., 2003] Thain, D., Tannenbaum, T., and Livny, M. (2003). Condor and the grid. *Grid computing: Making the global infrastructure a reality*, pages 299–335.

[Thaler et al., 2020] Thaler, J., Shin, W., Roberts, S., Rogers, J. H., and Rosedahl, T. (2020). Hybrid approach to hpc cluster telemetry and hardware log analytics. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7.

[Tikir et al., 2009] Tikir, M. M., Laurenzano, M. A., Carrington, L., and Snavely, A. (2009). Psins: An open-source event tracer and execution simulator for mpi applications. In *European Conference on Parallel Processing*, pages 135–148. Springer.

[Tuncer et al., 2017a] Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2017a). Diagnosing performance variations in hpc applications using machine learning. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D., editors, *High Performance Computing*, pages 355–373, Cham. Springer International Publishing.

[Tuncer et al., 2017b] Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M., and Coskun, A. K. (2017b). Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference*, pages 355–373. Springer.

[Venkata et al., 2009] Venkata, M. G., Bridges, P. G., and Widener, P. M. (2009). Using application communication characteristics to drive dynamic mpi reconfiguration. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–6.

[Vetter and Chambreau, 2014] Vetter, J. and Chambreau, C. (2014). mpiP: Lightweight, Scalable MPI Profiling. `http://mpip.sourceforge.net/`. [Online; accessed 10-Apr-2020].

[Videla and Williams, 2012] Videla, A. and Williams, J. J. (2012). *RabbitMQ in action: distributed messaging for everyone*. Manning.

[Wagner et al., 2017] Wagner, M., Mohr, S., Giménez, J., and Labarta, J. (2017). A structured approach to performance analysis. In *International Workshop on Parallel Tools for High Performance Computing*, pages 1–15. Springer.

[Webber, 2012] Webber, J. (2012). A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA. Association for Computing Machinery.

[Weidner et al., 2016a] Weidner, O., Atkinson, M., Barker, A., and Filgueira Vicente, R. (2016a). Rethinking high performance computing platforms: Challenges, opportunities and recommendations. In *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, DIDC '16, page 19–26, New York, NY, USA. Association for Computing Machinery.

[Weidner et al., 2016b] Weidner, O., Atkinson, M., Barker, A., and Filgueira Vicente, R. (2016b). Rethinking high-performance computing platforms: challenges, opportunities and recommendations. In *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, pages 19–26. ACM.

[Weidner et al., 2017] Weidner, O., Barker, A., and Atkinson, M. (2017). Seastar: A Comprehensive Framework for Telemetry Data in HPC Environments. In *the 7th International Workshop*, pages 1–8, New York, New York, USA. ACM Press.

[Weidner and Bidal, 2008] Weidner, O. and Bidal, J.-C. (2008). Shrimp farming on the grid. In *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*, pages 1–1.

[Wilde et al., 2014] Wilde, T., Auweter, A., and Shoukourian, H. (2014). The 4 pillar framework for energy efficient hpc data centers. *Computer Science-Research and Development*, 29(3-4):241–251.

[Yang et al., 2013] Yang, X., Zhou, Z., Wallace, S., Lan, Z., Tang, W., Coghlan, S., and Papka, M. E. (2013). Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11.

[Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper*, chapter SLURM: Simple Linux Utility for Resource Management, pages 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Zaharia et al., 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.