Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

# From Reality to Simulation: A Trace-Driven Closed-Loop Evaluation Methodology for Cloud Datacenters

**Author:** Guanghe Xie (2777594)

| | |
|---|---|
| *1st supervisor:* | Dr. Tiziano De Matteis |
| *daily supervisor:* | Dr. Matthijs Jansen |
| *2nd reader:* | Dr. Daniele Bonetta |


*A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science*

November 12, 2025

# Abstract

Trace-driven simulation is an effective tool for evaluating resource management policies in cloud datacenters, but the reliability of its conclusions is limited by fidelity gaps between the simulator and physical reality. This paper designs and implements a trace-driven, closed-loop evaluation framework that links the physical execution platform (Continuum) and the datacenter simulator (OpenDC) through a standardized Core Trace Schema (CTS) and a systematic validation-and-feedback loop.

The framework first uses CTS to convert real execution data into reproducible simulator inputs, and then calibrates and evaluates simulator fidelity via a input fidelity validation process that includes aligned metrics and quantitative criteria. In our experimental evaluation, the framework initially confirms the simulator's high fidelity under a baseline configuration. However, during subsequent design space exploration and output fidelity validation, we uncover a critical discrepancy: the simulator fails to correctly predict the performance and energy rankings of highly concurrent and oversubscribed strategies; the strategy it predicts as optimal performs the worst in reality.

This outcome highlights the necessity of the closed-loop approach. It shows that simulation-only evaluation can lead to incorrect optimization decisions, whereas output fidelity validation is a crucial step for revealing model limitations (e.g., missing concurrency-overhead models) and preventing the deployment of suboptimal strategies. The contribution of this work is to provide and validate an evaluation methodology that can significantly improve the reliability of decisions.

# Contents

# 1

# Introduction

In a digital and AI-driven economy, cloud datacenters have become critical infrastructure. They host massive online services and offline computation spanning e-commerce, video streaming, and machine learning (1). As scale and complexity increase, datacenters must continuously ensure high availability and high efficiency under multi tenancy, heterogeneous resources, and elastic loads (2), while "tail latency" is amplified in large-scale distributed systems such that a small number of slow requests can disproportionately degrade overall user experience and business revenue (3). These factors make performance and resource management decisions more difficult and more critical. Therefore, interpretable, reproducible, and cost-controllable methods for evaluation and optimization are essential for cloud datacenters.

Meanwhile, energy and sustainability are also key: the International Energy Agency (IEA) projects that datacenter electricity consumption will double to around 945 TWh by 2030, nearly 3% of global electricity consumption, with an average annual growth rate of about 15% during 2024–2030, far exceeding other electricity consuming sectors (4). As electricity demand rises, power/cooling Opex grows and capacity expansion faces grid and carbon constraints, raising planning costs and uncertainty. Hence, optimization must strike a finer balance among performance, energy efficiency, and capacity. Given the high cost and risk of infrastructure-level online experiments, it becomes necessary to run large-scale, reproducible "hypothesize–evaluate–compare" experiments off-line, enabled by simulation and digital twins, before deployment.

With respect to evaluation methods, real world experiments possess the highest external validity—generalizability to real sites, hardware, and workloads, and they best reflect real operational conditions; however, conducting large-scale experiments in real physical datacenters not only entails enormous financial and time costs, but also makes it difficult to

ensure identical experimental conditions, which in turn hinders the replication of results (5). Therefore, simulation has become a key means of studying cloud datacenter technologies: compared with real experiments, simulation can perform repeatable evaluations on large numbers of tasks and hosts within a short time, thereby supporting large-scale "hypothesis–experiment" analyses (6). In sum, evaluating new strategies through simulation is not only cost-effective but also yields results with good repeatability and scalability.

## 1.1 Context

Although simulation can efficiently explore a broad design space, its conclusions are highly sensitive to the input workload. There are two main approaches to generating workloads. The first is the Synthetic Workload, which involves artificially generating request sequences based on statistical models or preset parameters. This method is flexible and controllable, allowing workloads with specific characteristics to be created by adjusting model properties. However, its strong reliance on statistical distributions makes it difficult to accurately reproduce the arrival timing of events and the complex interdependencies among requests, which, to some extent, undermines the credibility and practical value of the simulation's conclusions (7) (8).

The second approach is Real-world Workload Traces, which are obtained by collecting data from various applications running on actual computing clusters. Because these traces originate from the real execution of applications, they can more accurately reflect dynamic characteristics such as workload correlation and burstiness. The drawback, however, is the greater difficulty of data collection (9).

Against this backdrop, Trace-driven Simulation has emerged, which specifically refers to using task logs (i.e., "traces") recorded from a real system as the driving input for a simulator. This method aims to preserve the authentic behavioral patterns and temporal dynamics of a workload to the greatest extent possible, thereby allowing for selective modeling and fine-grained analysis of specific system components (10). It enables researchers to reproduce real-world phenomena, such as job arrivals, resource consumption, and queue backlogs—within a simulation environment, and consequently, to evaluate scheduling and resource management strategies with greater precision.

The foundational data supporting this direction has also matured. From Google's large scale cluster traces to Alibaba's co-located datacenter workloads that mix online services with batch processing, these public datasets reveal real statistical properties of arrival patterns, service times, priorities, and colocation interference, which are of core value for

research on scheduling and resource management (11) (12). They have not only changed how researchers understand datacenter behavior but also provided reusable corpora and methodological references for trace driven evaluation, thereby advancing simulation and comparative studies that are closer to actual practice.

Although trace-driven simulation is a promising technique, if we want to use it to simulate a specific application and explore its scheduling and resource management strategies, we must collect the trace of the application running on a physical execution platform and then replay and assess its fidelity in a suitable simulation platform. This requires the integration of two distinct environments: a execution platform to generate authentic traces and a simulation platform to replay them. The core of our research is to design and implement a framework that bridges these two worlds, enabling a verifiable workflow from reality to simulation and back.

Through this process, we can rapidly explore a large set of policy configurations in the simulation platform to identify candidates that better meet the application's requirements, and subsequently reproduce and validate them on the execution platform.

## 1.2 Problem Statement

While integrating execution platforms with simulators to create a closed-loop evaluation workflow holds great promise, its practical implementation involves several key considerations. A central consideration is the interface used to translate data between the two environments.

Although many simulators provide dedicated parsers for well-known public datasets (such as the Google and Alibaba cluster traces) (13) (14), a more general interface is lacking when it comes to processing custom, real-world experimental data. Furthermore, the translation process itself involves a conceptual challenge: ensuring semantic consistency. This refers to the task of mapping low-level system metrics to the high-level abstractions used by simulators. In this process, it is crucial to preserve key scheduling semantics and resource states, such as precise timestamps and resource usage values. Any potential loss or simplification of information may impact how well the simulation reflects the real experiment, which in turn affects the confidence with which we can apply simulation-derived recommendations to real-world systems. Therefore, developing a more systematic interface is a vital step in designing this closed-loop architecture.

Solving the interface challenge is a necessary first step, but it immediately exposes a second, more profound problem: the lack of systematic fidelity validation and feedback.

While simulators are often validated, this is typically done only against large-scale public traces. What is lacking is a methodology for establishing a verifiable, closed-loop connection with a specific, reproducible physical execution platform, which involves an integrated and iterative process.

Without this tight feedback loop, the fidelity of the simulation for the specific platform and application under study cannot be verified. This creates a critical disconnect: performance improvements reported in simulation may not be realized in actual deployment. This is not necessarily because the simulation model is fundamentally flawed, but because uncalibrated, subtle differences between the abstract model and the physical environment lead to different outcomes. The conceptual challenge here has two parts: first, defining rigorous criteria for equivalence and metric alignment between the two domains; and second, developing a controlled calibration process to systematically reduce any observed discrepancies. The absence of such a closed-loop methodology is a major barrier to building trust in simulation results.

These interface and fidelity validation challenges lead to a central question: what is the measurable impact and practical value of a fully integrated physical/digital-twin framework? Although the concept of such a closed-loop system is appealing, its effectiveness in real-world scenarios has not yet been systematically evaluated. Therefore, the core problem is not just building the framework, but proving its worth. The conceptual challenge lies in evaluating the framework's performance across two key dimensions. The first is predictive accuracy, which must be defined with clear metrics such as trend consistency and bounded error. The second, and more critical, is its decision-support value: can the framework consistently guide operators to make superior management decisions regarding performance-energy relationship? Without a quantitative answer to this question, the proposed framework would remain a promising but unproven methodology.

## 1.3 Research Questions

To address the challenges of interfacing, fidelity validation, and utility assessment outlined in the previous section, our research is guided by three core questions. These questions systematically deconstruct the overall problem into parts. The first question targets the fundamental challenge of creating a standardized interface between real and simulated worlds. The second focuses on establishing a feedback loop for systematic input fidelity validation and calibration. Finally, the third seeks to quantify the end-to-end impact

and practical value of the resulting framework. Together, they implement the design and evaluation of the closed-loop methodology proposed in this thesis.

**RQ1**

**How can we design a system, centered around a Core Trace Schema (CTS), for systematically bridging the gap between raw execution data from a execution platform and the structured inputs required by a simulator?**

This question tackles the challenge of creating a standardized interface to ensure that real-world experimental traces can be faithfully replayed in a simulation. The conceptual challenge lies in defining an interface that is expressive enough to capture critical system dynamics while remaining general enough to be platform-agnostic. Our approach is to address this by designing a Core Trace Schema (CTS), a formal specification of the essential data fields and temporal semantics of a trace. Answering this question thus involves designing this schema and the corresponding data transformation pipeline, with the goal of enabling reproducible and portable simulation studies.

**RQ2**

**How can we design a input fidelity validation and feedback loop between the execution platform and the simulator that enables validation and rapid design space exploration?**

This question addresses the challenge of ensuring a simulation's fidelity to its physical counterpart. The conceptual difficulty lies in moving beyond one-off fidelity assessment to create a systematic, iterative feedback loop. Our approach is to design a reproducible, closed-loop methodology that involves trace-aligned replay, discrepancy analysis, and iterative model calibration. Answering this question involves defining the criteria for consistency and the process for controlled calibration, with the goal of making simulation not only verifiable but also a truly practical tool for system optimization.

**RQ3**

**What is the impact of the proposed physical/digital-twin methodology on the accuracy and decision-support value of performance and energy evaluations in a cloud datacenter scenario?**

This question evaluates the end-to-end effectiveness of the proposed methodology. The primary challenge is to move from concept to quantitative evidence of its value in a re-

alistic scenario. Our approach is to conduct a case study focused on performance-energy relationship in a datacenter. We will use the framework to first identify optimal policy configurations in simulation and then perform targeted output fidelity validation on the execution platform. Answering this question involves assessing the framework's predictive accuracy and, crucially, its ability to consistently guide correct strategy selection, thereby demonstrating its practical value as an evidence-backed evaluation tool.

## 1.4 Research Methodology

The following research methodologies are applied to answer the above research questions systematically:

### 1.4.1 (M1) Trace Contract Design and Abstraction

To address RQ1, we designed the Core Trace Schema (CTS) through a structured, multi-stage methodology. The primary goal was to define a standardized and platform-agnostic interface. This required a deep understanding of both the data that could be realistically collected from physical systems and the data required by simulators.

Therefore, the design of the CTS was preceded by a systematic, criteria-based selection of our architecture's foundational tools. This process involved two key steps:

1. **State-of-the-Art Analysis:** We began by conducting a survey of the landscape of modern execution platforms and datacenter simulators. The goal of this survey was not to select tools immediately, but to identify the common patterns, capabilities, and data models used in the field.

2. **Requirements Elicitation:** Based on this analysis, we elicited a set of high-level requirements. For the execution platform, these centered on reproducibility, controllability, and observability (e.g., energy monitoring). For the simulator, requirements focused on trace-driven capabilities, model extensibility, and data export features to support a feedback loop.

These requirements served as a set of criteria for a comparative analysis, which guided our final selection of the execution platform and simulator (detailed in Chapter 2). After establishing specific constraints of the chosen tools, we then defined the CTS through an iterative design process. This process focused on identifying the core set of fields and semantic invariants necessary to bridge the two platforms while preserving the fidelity needed for fidelity assessment. The resulting CTS specification is presented in Chapter 3.

### 1.4.2 (M2) Input Fidelity Validation–Feedback Loop Methodology

To answer RQ2, we designed the input fidelity validation-feedback loop based on a methodology that prioritizes three key objectives. First, to ensure reproducibility, our method establishes a strict baseline alignment between the physical and simulated systems. Second, to enable credible fidelity assessment, the method relies on direct comparison using the standardized CTS and pre-declared consistency metrics. Finally, to achieve model fidelity, the method includes an iterative calibration process, where observed discrepancies are systematically reduced through a feedback loop. This approach ensures the resulting input fidelity validation process is robust and mitigates researcher bias.

### 1.4.3 (M3) Experimental Design and Evaluation Plan

To quantitatively answer RQ3, our methodology employs a two-stage, resource-efficient evaluation strategy. This approach allows us to rigorously assess the framework's practical value without the prohibitive cost of exhaustive physical testing.

The first stage consists of a broad, simulation-based Design Space Exploration (DSE). Here, we use the simulator's speed to evaluate a wide range of policy configurations under controlled conditions, identifying a small set of the most promising candidates. The second stage then involves targeted output fidelity validation, where only these top-ranked candidates are executed and measured on the execution platform. The framework's overall effectiveness is then assessed against our two criteria of predictive accuracy and decision-support value, determined by the ranking consistency between the simulated and output validation results. This two-stage approach provides a pragmatic and robust method for evaluating the end-to-end value of our framework.

### 1.4.4 (M4) Open Sourcing and Reproducibility

We organize and release all datasets, conversion scripts, simulation configurations, and code examples from this study to achieve reproducibility. The appendix or supplementary materials provide an end to end guide from real data collection to simulation driving and then to output fidelity validation, facilitating reference and extension by subsequent researchers.

## 1.5 Thesis Contributions

The main contributions of this thesis are as follows:

- **(Contribution C1, Design)** The formalization of a Core Trace Schema (CTS), a standardized and platform-agnostic specification for representing real-world execution traces. This contribution includes the design of a systematic transformation pipeline that converts raw, partially-observed system data into the CTS format while preserving critical temporal and semantic invariants.

- **(C2, Design)** The design of a methodology for closed-loop input fidelity validation, founded on principles of reproducibility, empirical comparison, and iterative refinement. This methodology provides a systematic process for assessing the input fidelity of simulator against its physical counterpart by using trace-aligned replay, discrepancy analysis, and controlled parameter calibration, transforming simulation into a verifiable and trustworthy tool.

- **(C3, experimental)** The design and application of a two-stage evaluation strategy for quantitatively assessing the end-to-end effectiveness of a physical/digital-twin methodology. This contribution demonstrates how to efficiently measure both predictive accuracy (via trend consistency and bounded error) and decision-support value (via ranking consistency), providing an evidence-backed confirmation of the methodology's practical utility.

## 1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

## 1.7 Thesis Structure

This subsection presents the structure of this thesis.

First, Chapter 1 is the introduction. It outlines the research background, states the problem, and then introduces the research questions (RQs), research methodology, and main contributions. Chapter 2 presents the background knowledge required to understand our work. It covers the basics of trace-driven simulation, the motivation for tool selection, and the specific architectures of the chosen platforms—the OpenDC simulator and the Continuum framework. Chapter 3 explains the complete architectural design of the evaluation framework. It details the design objectives, the specification of the Core Trace Schema (CTS), the data-processing pipeline, the input fidelity validation-and-feedback loop, and

the strategy for simulation-driven evaluation. Next, Chapter 4 describes the concrete implementation of this design. It covers the technology stack, the data-collection strategy, the pipeline that converts raw data into CTS artifacts and OpenDC inputs, and the implementation of the input fidelity validation engine, including aligned metric computation and consistency criteria. Then, in Chapter 5, we conduct a comprehensive experimental evaluation. The evaluation first compares simulated results with real results to perform rigorous input fidelity validation. Building on this, we use the simulator for design space exploration to identify optimized strategies. Finally, the top-ranked strategies are deployed back on the Continuum platform for output fidelity validation to assess the framework's end-to-end decision-support value. In Chapter 6, we review related work and survey research concerning trace-driven simulation and simulation fidelity validation. Finally, Chapter 7 concludes the thesis, answers the research questions (RQs), highlights the core findings and contributions, and discusses limitations and directions for future work.

# 2

# Background and Motivation

This chapter establishes the technical foundations of this thesis, organized to present a clear progression from general concepts to the specific tools selected for our framework.

We begin in Section 2.1 by defining trace-driven simulation, the core methodology central to our work. Following this, in Section 2.2, we introduce the conceptual roles of the two key components of our architecture: the execution platform, which we use as a "controlled reference instrument," and the simulation platform, as a "cost-effective exploration engine."

Building on these definitions, Section 2.3 details the specific technical requirements for these tools. Crucially, it presents a comparative analysis of alternative solutions, systematically motivating our selection of Continuum and OpenDC. Finally, Sections 2.4 and 2.5 provide detailed overviews of the Continuum framework and the OpenDC simulator, respectively, focusing on the specific capabilities that are used in our design. Understanding these elements, their roles, and the reasons for their selection is essential for following the design and implementation details presented in subsequent chapters.

## 2.1 Trace-driven simulation

Trace-driven simulation is understood here as using task/job events and resource-usage data recorded from a real system (traces) as inputs to the simulator, and replaying them in a discrete-event model according to their original timing. Compared with purely synthetic workloads, this approach preserves key behavioral semantics from reality, such as submission/start/finish times and workload burstiness, so that simulation results are closer to real systems and more transferable.

In this thesis, traces do not serve as a independent proof of simulation fidelity but rather as a common baseline between the execution platform and the simulator. The same trace

can produce observable throughput, latency, and energy series on the execution platform and can be stably replayed and evaluated in the simulator, enabling aligned comparison, error breakdown, and model calibration. To reduce information loss and conversion overhead, we focus on the core field set required to drive replay and its time semantics, and formalize this as the Core Trace Schema (CTS). To practically apply this trace-driven methodology, it is essential to select a suitable execution platform to generate traces and a simulator to replay them. The following section details the specific requirements for these components and our reasons for selecting them.

## 2.2 Foundational Components of the Architecture

The design of our closed-loop evaluation framework requires the careful integration of two foundational components: an execution platform and a datacenter simulator. Before detailing the specific technical requirements for these tools, it is essential to first define their roles within our research.

- **The Execution Platform as a "Controlled Reference Instrument":** In this framework, the execution platform's primary role is to serve as a controlled environment for generating high-fidelity, reproducible traces. It must provide strong isolation between experimental runs and offer fine-grained observability of performance and energy metrics. This ensures that the collected data serves as a trustworthy "ground-truth" baseline for the simulator's input fidelity validation.

- **The Simulation Platform as a "Cost-Effective Exploration Engine":** The simulator, in contrast, functions as the framework's exploration engine. Its purpose is to ingest the real-world traces and enable the rapid, large-scale evaluation of numerous "what-if" scenarios —such as alternative scheduling policies or resource configurations—at a fraction of the cost and time of physical experiments. This requires a modular design for extensibility and fidelity-assessment-oriented features to support the feedback loop.

Having defined these roles, we can now derive a set of specific requirements to guide our selection of the most appropriate tools.

## 2.3 Tooling Requirements and Selection Rationale

The design of our closed-loop evaluation framework requires the careful integration of two foundational components: a execution platform and a datacenter simulator. The selection of these tools was not arbitrary but followed a systematic, criteria-based process designed to ensure they could jointly meet the research objectives outlined in Chapter 1. This section details the requirements we established for each component and justifies our selection of Continuum and OpenDC.

### 2.3.1 Requirements and Selection of the Execution Platform

To function as a controlled "reference instrument," the execution platform must satisfy three primary requirements:

- Reproducibility and Isolation: The platform must provide strong isolation between experimental runs to ensure results are repeatable. We favor solutions based on virtual machines (VMs), where the entire software stack, including the operating system and kernel versions, can be fixed, providing a maximally stable baseline for both execution and subsequent comparison with a simulator.

- Controllability and Usability: The platform must be easily configurable to represent various datacenter topologies. A declarative configuration method is preferred for simplifying experimental setup and ensuring auditability.

- Observability: The platform must allow for the integration of fine-grained monitoring, particularly for energy consumption. This requires a clear pathway to collect host-level hardware counter data (e.g., from Intel RAPL) and align it with application performance metrics.

To select the most suitable platform, we compared several state-of-the-art alternatives. The comparison, based on their official documentation and related publications, is summarized in Table 2.1.

Our analysis reveals that while all platforms provide robust mechanisms for reproducibility, their underlying architectures lead to important trade-offs. The first key differentiator is the isolation primitive. For our goal of creating a ground-truth baseline, a VM-based approach as used by Continuum and MockFog is preferable to a container-based one (Fogify, EmuFog). VMs provide full virtualization of both hardware and the operating system,

**Table 2.1:** Comparison of Execution Platform Alternatives.

| **Criterion** | **Continuum (15)** | **Fogify (16)** | **EmuFog (17)** | **MockFog 2.0 (18)** |
|---|---|---|---|---|
| Isolation Primitive | Virtual Machines (VM) | Containers (Docker) | Containers (Docker) | VMs & Containers |
| Reproducibility | High | High | High | High |
| Topology Definition | Declarative (Single file) | Declarative (Docker Compose) | Script-based (Python) | Declarative |
| Energy Monitoring Pathway | Host-level (VM attribution) | Host-level (Container attribution) | Host-level (Container attribution) | Host-level (VM attribution) |

ensuring that experiments are shielded from variations in the host's kernel and library versions, thus offering a higher degree of environmental control.

Regarding energy monitoring, our analysis confirms that no platform provides a native solution; all require an external, host-level collection mechanism. The pathway—collecting data via tools that read hardware counters like Intel RAPL and attributing it to either VM or container processes—is conceptually similar across all platforms. Therefore, this criterion is not a primary differentiator for selection.

The decisive factor for our selection is the combination of strong isolation and configuration simplicity. Continuum's use of strong VM-based isolation meets our requirement for a stable baseline, while its simple, single-file declarative configuration for defining both VM specifications and network topology offers superior usability and auditability for our specific use case of automated benchmarking. This combination makes it the most suitable "reference instrument" for generating the high-fidelity traces required by our study.

### 2.3.2 Requirements and Selection of the Simulation Platform

To function as a cost-effective "exploration engine," the datacenter simulator must fulfill three key requirements:

- Trace-driven Core: The simulator must be fundamentally designed to be driven by real-world execution traces. This native replay capability is essential for ensuring a direct and faithful comparison with the physical baseline, minimizing the need for complex, custom-built adapters.

**Table 2.2:** Comparison of Datacenter Simulator Alternatives.

| **Criterion** | **OpenDC 2.0 (6)** | **CloudSim (19)** | **SimGrid (20)** | **EdgeCloudSim (21)** | **iFogSim2 (22)** |
|---|---|---|---|---|---|
| Primary Focus | Datacenter Performance & Energy | General Cloud Services | Distributed Applications & HPC | Edge Computing | IoT & Fog Computing |
| Trace-driven Support | Native | Via extension | Native | Via extension | Via extension |
| Policy Extensibility | Pluggable Architecture | Inheritance & Overwriting | API & Model Replacement | Modular (Factory) | Inheritance & Overwriting |
| Data Export for Validation | High (Engineered for reproducibility) | High (CSV) | High (Trace analysis tools) | High (CSV) | Medium (Quantitative metrics) |

- Extensibility and Flexibility: The platform must feature a modular design, allowing for the straightforward implementation and evaluation of custom scheduling policies and energy models, which is central to our design space exploration.

- Fidelity-Validation-Oriented Features: The simulator must be able to export detailed, time-series data of its internal state (e.g., task events, resource usage, power draw) in an accessible format. This is a prerequisite for the quantitative comparison and calibration central to our closed-loop methodology.

Based on these criteria, we surveyed several well-established datacenter simulators. Our comparative analysis is summarized in Table 2.2.

The analysis shows that while all simulators are powerful tools, their core design philosophies make them suitable for different research goals. For our study, the most critical requirement is native support for trace-driven simulation. As indicated in the comparison, only OpenDC and SimGrid are natively designed for this "replay" capability. Other simulators like CloudSim and EdgeCloudSim are primarily driven by synthetic workload generators and would require significant adaptation to ingest our custom traces, potentially introducing inconsistencies.

Between the two natively trace-driven simulators, the choice is determined by their primary focus. SimGrid is a highly capable framework for modeling distributed applications and network protocols. However, our research questions are centered on datacenter-level resource management, specifically the trade-offs between performance and energy. OpenDC's focus on "Cloud Datacenters," with built-in concepts for virtualization, batch processing, and energy consumption, is much more closely aligned with our research domain.

Finally, OpenDC's design for extensibility and fidelity validation meets our remaining requirements perfectly. Its modular, pluggable architecture facilitates the rapid testing of different scheduling policies, and its data export features are explicitly engineered to support reproducible experiments, which is the cornerstone of our closed-loop input fidelity validation approach.

Therefore, OpenDC was selected because its combination of native trace-driven support, a specific focus on datacenter performance and energy, and an architecture designed for extensibility and validation makes it the most suitable simulation platform for this research.

## 2.4 OpenDC Simulator

OpenDC is a discrete event simulation platform for cloud datacenters that emphasizes usability, extensibility, and reproducibility. Its core comprises three parts: pluggable resource and scheduling/placement models, trace-based workload ingestion and replay, and results instrumentation/export. The design allows scheduling and placement policies to be swapped or combined without modifying the simulation kernel, and enables batch execution and reproduction of experiments under a unified interface. Consequently, it supports policy comparison and design space exploration on the same trace and under equivalent resource configurations (6). These properties align closely with the input fidelity validation and feedback loop proposed in this thesis, which consists of replay, alignment, calibration, and output fidelity validation: OpenDC reliably accepts the Core Trace Schema (CTS) field set, replays submission/start/finish events according to their original time semantics, and exports key metric series (throughput, completion time, energy) for trend and bounded-error comparisons against physical observations.

At the input and modeling level, OpenDC includes native support for trace-driven workload loading and event progression, and provides resource hierarchies that span host / rack / cluster along with queue-based scheduling interfaces, covering the CPU-task focus of this thesis baseline. It also offers an energy model and metrics pipeline capable of producing energy time series and efficiency indicators, directly serving performance-energy trade-off

analysis and alignment. For experimental organization, OpenDC supports the declarative specification of parameters and repetitions, fixes random seeds, and runs experiments in batches, thus reducing degrees of freedom and improving reproducibility, consistent with the variance control and output fidelity validation procedures of this thesis (6).

It is worth noting that our use of OpenDC intentionally constrains modeling granularity to avoid semantic mismatch with the execution side: we do not rely on fine-grained network and storage modeling, but focus the simulation on arrival–queueing–service processes and CPU resource-contention semantics that align with the traces. For events treated as approximately instantaneous (e.g., start / termination), we compensate for execution–simulation differences via a closed loop of error breakdown, parameter calibration, and small-sample output fidelity validation. This preserves replayability and experimental efficiency while meeting the accuracy criterion of this thesis of 'trend consistency + bounded error.'

Finally, OpenDC's open source implementation continues to evolve and maintains solid engineering readiness: the official repository exposes modules for the simulation core, trace handling, and experiment orchestration. This engineering status facilitates the reproducibility and extensibility required by our study (23).

## 2.5 Continuum Framework

Continuum is an automated deployment and benchmarking framework for the compute continuum (cloud–edge–endpoint). Its core goal is to reproduce end-to-end experiments—from infrastructure to software stack to application benchmarks—with minimal configuration. Using virtual machines (VMs) as the fundamental abstraction, it can rapidly construct controllable multi-tier environments locally (QEMU/KVM) or in the cloud (e.g., GCP), and allows per-tier (cloud/edge/endpoint) specification of the number of devices and per-VM limits on CPU, memory, and storage, as well as inter- and intra-tier network latency and bandwidth. This makes it possible to approximate target deployments on commodity hardware and to reproduce the resource and network conditions required by experiments. For our purposes, this capability enables us to build a execution side baseline under "the same trace + equivalent resources," providing a reproducible reference for subsequent alignment against the simulation side (15).

# 3

# Design of the Trace Schema and Processing Framework

With the foundational tools selected, this chapter presents the architectural blueprint for a framework designed to systematically bridge the physical and simulated worlds. A principled architecture is required to ensure the resulting workflow is reproducible, verifiable, and trustworthy.

We will detail the core pillars of this design: the formal trace schema, the data processing pipeline, and the methodology for the closed-loop input fidelity validation and feedback.

## 3.1 Design Goal and Requirements

The primary design goal is to architect a standardized and reproducible workflow, centered on a formal Trace Schema and the Processing Framework detailed in this chapter, that creates a verifiable bridge between the physical execution platform and the simulator. To guide this architectural design, we first establish a set of functional and non-functional requirements derived from the research objectives outlined in Chapter 1.

### 3.1.1 Functional Requirements

The functional requirements (FRs) specify the essential behaviors and capabilities the framework must provide to achieve the objectives of this study.

- **FR1: Trace Collection.** The designed system must have the capability to collect raw, multi-source execution data and energy consumption data from a physical environment to serve as the basis for simulation.

- **FR2: Trace Standardization.** The designed system must be able to transform raw traces into a standardized, platform-agnostic representation to ensure portability and reproducibility.

- **FR3: Workload Generation.** The designed system must be capable of converting a standardized trace into a simulator-specific workload and an equivalent resource configuration to enable a fair comparison.

- **FR4: Aligned Replay.** The workflow must support the reliable replay of workloads in the simulator and the export of the key performance and energy metrics that are semantically aligned with their physical counterparts for direct comparison.

- **FR5: Model Calibration.** The designed system must provide a mechanism to systematically tune simulator parameters based on observed discrepancies, in order to improve the model's fidelity.

- **FR6: Simulation-Driven Exploration.** The workflow must be able to use the validated simulator to explore a design space of different policies and rank them based on predefined performance and energy criteria..

- **FR7: Output Fidelity Validation Support.** The workflow must support the redeployment of simulator-identified candidate policies back to the execution platform to verify their effectiveness in a real-world context.

### 3.1.2 Non-Functional Requirements

The non-functional requirements (NFRs) capture the framework's quality attributes.

- **NFR1: Accuracy**. The workflow must produce simulation results that are demonstrably accurate. Accuracy is defined as the combination of trend consistency (i.e., the simulation and physical measurements follow similar patterns over time) and bounded error (i.e., the numerical difference between them remains within predeclared thresholds).

- **NFR2: Reproducibility**. The workflow must produce statistically consistent results for the same experiment. This requires that all sources of non-determinism are controlled, for example by fixing random seeds, configurations, and software versions for each experimental run.

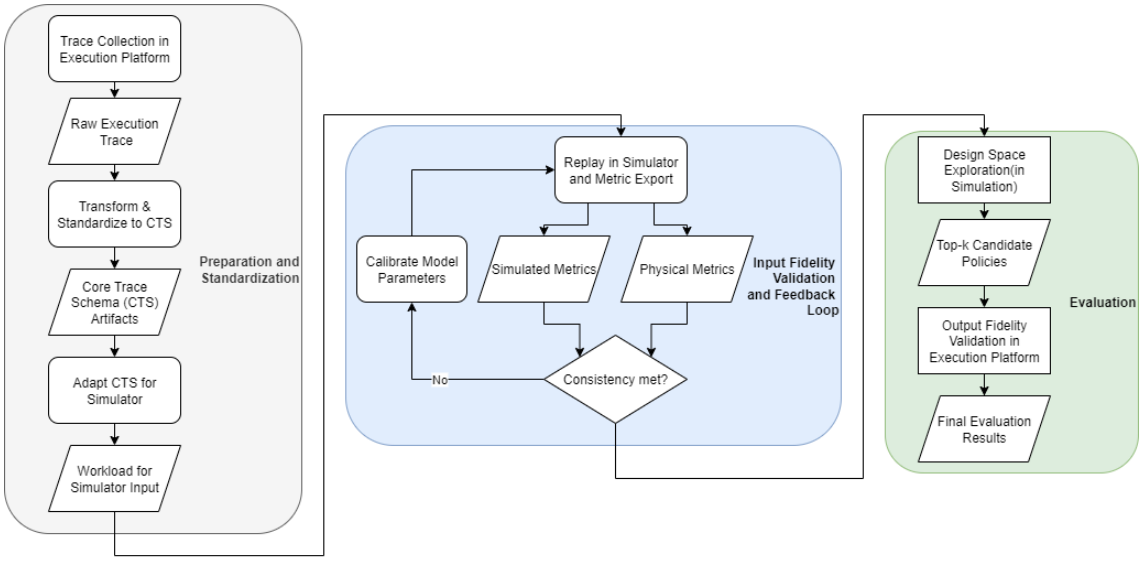


**Figure 3.1:** Architecture Overview.

- **NFR3: Semantic Alignment**. All metrics reported by the physical and simulated platforms must be semantically aligned. This ensures a true "apples-to-apples" comparison by enforcing consistency in metric definitions, units, scopes, and aggregation windows.

- **NFR4: Low Overhead**. The data collection process on the execution platform must have a negligible performance impact on the system under measurement. The trace processing and simulation workload generation should not interfere with the physical experiment.

## 3.2 Design Overview

To satisfy the requirements outlined in the previous section, our architecture is designed around two core principles: modularity and semantic fidelity. Modularity ensures that components can be independently developed and extended, which is critical for a research framework. Semantic fidelity means that the essential characteristics of the real-world workload are preserved throughout the process so that final results are trustworthy.

The architecture implements a three-stage workflow, as visualized in Figure 3.1. This structure provides a clear separation of concerns that directly maps to the functional requirements:

1. **Preparation and Standardization:** This initial stage addresses the requirements for data ingestion and standardization (FR1–FR3). It focuses on reliably capturing real-world behavior from the execution platform and translating it into a standardized, reusable format—the Core Trace Schema (CTS). In the meantime, energy consumption data will also be collected. This stage serves as the foundation for the entire workflow, ensuring that the data entering the loop is consistent and of high quality.

2. **Input Fidelity Validation and Feedback Loop:** This stage fulfills the need for model fidelity and calibration (FR4–FR5). It forms the core of the architecture's closed-loop nature, ensuring the simulator is a trustworthy representation of its physical counterpart via an iterative process of trace-aligned replay, quantitative comparison against pre-declared criteria, and controlled model refinement.

3. **Evaluation:** This final stage provides capabilities for policy exploration and output fidelity validation (FR6–FR7). It uses the validated simulator to generate actionable insights on performance-energy trade-offs by rapidly exploring a large design space. The top-ranked policies identified in simulation are then deployed back to the execution platform for targeted output fidelity validation, closing the loop and confirming their real-world effectiveness.

To realize this three-stage workflow, our design is composed of several key logical components. The Data Processing Pipeline implements the Preparation and Standardization stage. It is responsible for collecting raw data from the physical environment and transforming it into two distinct outputs: (1) the Core Trace Schema (CTS) to define the workload for replay, and (2) a time-aligned energy trace for later fidelity validation. Subsequently, the Validation Engine implements the Input Fidelity Validation and Feedback Loop by consuming these CTS artifacts to manage consistency checks and calibrate the simulator. Finally, the Evaluation stage is an application of the entire validated system, leveraging the now-trusted simulator to perform design space exploration and output fidelity validation.

The subsequent sections of this chapter will provide detailed specifications for these components and the CTS itself.

## 3.3 Core Trace Schema (CTS) Design

The Core Trace Schema (CTS) is the central data interface of our architecture, designed to be the standardized language spoken between the physical and simulated worlds. Its design

is not arbitrary but is derived from a systematic analysis of the inherent conceptual and data-format differences between these two domains. The primary challenge is to bridge the "semantic gap" between the data that can be realistically collected from a physical execution platform and the structured inputs required by a discrete-event datacenter simulator.

To ensure our design is both robust and practical, it is informed by an analysis of the typical capabilities and requirements of these two classes of systems.

### 3.3.1 Analysis of the Data Source: Characteristics of Physical Execution Platforms

Physical execution platforms, which serve as the source of our ground-truth data, typically provide two distinct categories of information about a running workload:

- **High-Precision, Discrete Lifecycle Events:** These are high-level events that define a task's existence and its interaction with a scheduler. Key events include a task's submission, start of execution, and completion. These timestamps can often be captured with high precision at the application layer or through wrapper scripts.
- **Continuous, Cumulative System-Level Metrics:** These are fine-grained measurements of computational resources consumed by a task during its execution. They are typically sourced from the operating system and have two key characteristics: they are cumulative (e.g., total CPU time consumed since process start) and are sampled at discrete intervals. This raw, cumulative data cannot be directly used to model a task's time-varying resource demand in a simulation.

### 3.3.2 Analysis of the Data Sink: Requirements of Datacenter Simulators

Discrete-event simulators, which act as the target for our traces, require highly structured and abstracted inputs to drive their models. Conceptually, a typical datacenter simulator needs:

- **High-Level Task Definitions:** A representation of each task's essential attributes, such as its submission time, dependencies, and a high-level profile of its resource requirements.
- **A Time-Discretized Resource Demand Profile:** Because discrete-event simulators model time as a series of events rather than as a continuous flow, they cannot

natively represent continuously changing resource usage. Instead, they must approximate a task's execution as a sequence of time intervals, during each of which the task is assumed to consume resources at a piecewise-constant rate.

The core semantic gap is clear: the physical world provides continuous, cumulative usage data, while the simulated world requires a discrete, incremental representation of resource demand.

### 3.3.3 Deriving the CTS Design to Bridge the Semantic Gap

The design of the CTS directly addresses this semantic gap by logically decoupling a workload's execution into three components, each serving as a necessary step in the transformation pipeline.

- **Task Lifecycle Events:** This component of the schema is designed to capture the high-precision, discrete events from the execution platform. This directly satisfies the simulator's need for high-level task definitions and provides the foundational data for analyzing scheduling behavior.
- **Time-Series Resource Usage:** This component is the key to bridging the core semantic gap. Its design is based on representing resource consumption as a sequence of time-stamped samples collected from the OS. The fundamental design principle here is that this series of cumulative measurements can be transformed via post-processing and differencing into the incremental resource usage within each sampling window. This calculated rate of consumption is the essential precursor required to construct the time-discretized resource demand profile needed by the simulator.
- **Static Environment Description:** This component is designed to describe the static properties of the physical infrastructure where the trace was collected. This intentionally decouples the dynamic workload from the static environment, allowing the same trace to be replayed on different simulated infrastructures while ensuring the input fidelity validation is performed on an equivalent configuration.

In conclusion, the three-part structure of the CTS is a direct result of our analysis. It is designed to be expressive enough to capture the essential characteristics of a real-world execution while being structured in a way that facilitates the necessary transformation into a format suitable for faithful, trace-driven simulation. The precise implementation of this design is detailed in Chapter 4.

## 3.4 The Data Processing Pipeline Design

The Data Processing Pipeline is the component responsible for implementing the "Preparation and Standardization" stage of our workflow. Its design is engineered to be a robust and auditable process that transforms raw, heterogeneous data from the physical environment into the clean, standardized CTS format required for simulation. The pipeline is logically divided into two major stages: first, generating the CTS from raw observations, and second, adapting the CTS for a specific simulator.

### 3.4.1 Generating the CTS from Raw Observations

The primary design challenge in this stage is to reliably converge data from multiple, disparate sources into a single, coherent representation of the experiment. Our design structures the process into three logical steps:

1. **Multi-Level Data Sourcing:** The design specifies a collection strategy that gathers data from three logical layers to construct a complete picture of system behavior. The Application Layer is the designated source for high-precision task lifecycle events. The Operating System Layer provides dynamic, process-level resource consumption metrics and static node specifications. The Hardware/Virtualization Layer sources direct energy-consumption measurements for fidelity validation. While the overall design of the pipeline is intended to be general-purpose, the initial Data Collection stage is inherently application-aware. The specific mechanisms for capturing high-precision lifecycle events, in particular, must be adapted to the type of application being traced.

2. **Core Data Transformation:** The design centers on key methods to bridge the semantic gap between continuous physical-world measurements and the discrete-event nature of simulation. A core design decision is *Windowing and Discretization*, converting cumulative, continuous metrics (e.g., CPU time) into a sequence of discrete workload fragments. Another crucial aspect is *Multi-Source Time Alignment*, which mandates synchronizing all data streams to a unified clock to preserve causal relationships.

3. **Quality Assurance and Auditing:** To ensure the process is trustworthy, the design incorporates an automated quality-assurance step. This validates generated CTS

artifacts for logical integrity (e.g., timestamp invariants and cross-file referential integrity) and produces an auditable report, ensuring the transformation is transparent and reproducible.

### 3.4.2 Adapting the CTS for a Target Simulator

This final stage of the pipeline acts as a "translator," converting standardized CTS artifacts into a proprietary, simulator-specific format. The key goal is to perform this translation losslessly while ensuring semantic equivalence.

- **Format and Structure Mapping:** A mapping module translates CTS conceptual entities (e.g., a process execution) into the simulator's concrete entities (e.g., a "task" object). A key decision is to use the `nodes.json` artifact from the CTS as the single source of truth for generating a consistent simulation topology, cleanly decoupling the dynamic workload from the static environment.

- **Resource Semantics Modeling:** Simulators often require high-level resource requests (e.g., "this task needs 2 cores"), whereas traces contain low-level usage data. We incorporate a statistical modeling module that analyzes time-series usage to infer a robust and representative resource-capacity profile for each task, filtering noise while preserving key performance dynamics.

The specific tools, commands, and statistical models used to implement this pipeline are detailed in Chapter 4. With a standardized and simulator-ready workload now available, the next component of our design is the engine that uses this workload to validate the simulation model itself.

## 3.5 Input Fidelity Validation and Feedback Loop Design

This section details the design of the Input Fidelity Validation Engine, the component responsible for implementing the "Input Fidelity Validation and Feedback Loop" stage of our workflow. The design of this engine is centered around a core input fidelity validation workflow, which specifies how we conduct reproducible and quantifiable consistency checks between the simulator and its physical counterpart.

The workflow is designed to proceed as follows:

1. **Establish a Physical Baseline:** First, the target application is executed on the execution platform under a specific configuration. The resulting performance and energy measurements from this run serve as the baseline ground truth.

2. **Generate Simulation Inputs:** Concurrently, the Data Processing Pipeline (as designed in Section 3.4) collects a trace from this baseline run and transforms it into the standardized CTS artifacts and simulator-specific inputs.

3. **Configure the Simulated Environment:** The simulator is configured with a topology and resource capacities that are equivalent to the physical baseline environment. The same policies (e.g., scheduling) are also applied.

4. **Execute the Simulated Run:** The simulator replays the workload derived from the physical trace.

5. **Perform Consistency Assessment:** Finally, the Input Fidelity Validation Engine compares the outputs from the simulated run against the ground truth measurements from the physical baseline. This comparison is guided by the aligned metric definitions and assessment criteria detailed in the following subsections.

The core objective of this workflow design is to rigorously evaluate the simulation model's fidelity and to provide a principled basis for subsequent parameter calibration if the consistency criteria are not met.

### 3.5.1 Aligned Metric Definitions

To comprehensively evaluate the model's fidelity, our design specifies a multi-dimensional set of Key Performance Indicators (KPIs) that provide a holistic view of system behavior, moving from high-level scheduling dynamics to low-level resource and energy patterns:

- **Queuing and Scheduling Behavior:** Focuses on the simulator's ability to reproduce queuing delays and resource contention experienced by tasks.

  **Core Metric: Task Wait Time**—the duration between a task's submission and the start of its execution. Faithful reproduction indicates the simulator's scheduling logic captures the dynamics of the physical system.

- **Macro-level Performance Progression:** Assesses whether the simulation captures overall execution progress and processing capacity.

**Core Metric: Cumulative Completion Curve** tracking completed tasks over time. Comparing curves evaluates whether the simulation exhibits the same total makespan and throughput characteristics as the execution platform.

- **Microscopic Resource Fidelity:** Evaluates the fidelity of trace transformation and replay at a fine-grained level.

  **Core Metric: Instantaneous CPU Usage**, which directly tests the conversion of CTS data into workload fragments by verifying that the intrinsic behavioral profile of each simulated task aligns with its real-world counterpart, independent of its absolute start time.

- **Energy Consumption Dynamics:** Assesses the fidelity of the simulator's energy model.

  **Core Metric: Total Power Draw Trend**—which focuses on evaluating the consistency of dynamic patterns throughout the experiment's lifecycle. The comparison emphasizes the "shape" and "synchronicity" of power-consumption curves rather than absolute values, verifying that workload-driven energy dynamics are reproduced without brittle hardware-energy calibration.

### 3.5.2 Consistency Assessment Criteria

For each metric dimension above, our design specifies corresponding quantitative assessment criteria to transform the abstract goal of "fidelity" into a concrete, measurable, and falsifiable objective. Each metric is paired with (i) an appropriate comparison methodology and (ii) a numerical threshold for determining pass/fail.

Methodologies are tailored to metric nature. For distributional metrics (e.g., Task Wait Time), we use quantile comparisons (e.g., p50, p95) and statistical tests (e.g., Kolmogorov–Smirnov) to compare overall shape. For time-series metrics such as performance progression and energy trends, we apply lag-robust comparison techniques combined with standard error metrics (e.g., RMSE, sMAPE) and correlation analysis (e.g., Pearson correlation) to assess both absolute error and trend similarity. For the energy trend, however, a simple global correlation is insufficient as it may mask inconsistencies during specific phases of the experiment. Therefore, our design employs a segmented assessment methodology. This approach partitions the total experiment duration into multiple segments and evaluates the trend correlation independently within each. This design decision ensures that our validation focuses on a more stringent criterion: sustained consistency across the entire

run, rather than just an overall average. The final judgment is based on the mean of the segmental correlation scores, ensuring the simulation's fidelity is robust and reliable throughout. Each comparison is evaluated against a pre-declared threshold to prevent post-hoc bias. The concrete tests, error metrics, and numerical thresholds are detailed in Chapter 4.

### 3.5.3 Parameter Calibration Method

The final component of our input fidelity validation-loop design is a systematic method for parameter calibration, providing a structured process to improve model fidelity when discrepancies are detected. The design is founded on a post-hoc, input-transformation-based principle. This approach ensures modularity and transparency by avoiding modifications to the simulation kernel, operating instead by systematically transforming the simulator's input artifacts before a re-run.

The design targets several categories of tunable parameters that correspond to common sources of discrepancy, implemented as scaling factors or offsets applied to the simulation's inputs:

**System-level Efficiency Factors:** To correct for macroscopic deviations in overall makespan or task processing speeds, the design specifies applying scaling factors to the simulator's topology definition (e.g., host processing speed) or to the workload's timing properties (e.g., task service durations).

**Fixed Timing Overheads:** To account for unmodeled, constant costs such as OS-level delays, the design includes the addition of a fixed time offset to the start of each task's execution within the workload trace.

**Resource Allocation Modifiers:** To align the effective throughput of the simulator with the physical system, the design allows for adjustments to the simulator's resource allocation policies via its configuration files.

The calibration process is designed as a systematic, operator-guided workflow. It is initiated when the consistency assessment returns a "FAIL". Based on which metric failed, a pre-defined "decision recipe" guides the operator to adjust the most relevant parameter category. This iterative loop of transforming inputs, re-running the simulation, and re-assessing the results continues until all consistency criteria are satisfied. To ensure reproducibility, the design mandates that all transformations are tracked and auditable.

## 3.6 Design for Simulation-Driven Evaluation

This section details the design of the final stage of our workflow: *Evaluation*. Having established a validated simulation model, this stage specifies the process for leveraging that model to generate and verify actionable insights. The design comprises two phases, directly corresponding to FR6 and FR7.

### 3.6.1 Design for Design Space Exploration

The first phase of the Evaluation stage is a systematic exploration process that exploits the speed and low cost of the validated simulator. The design follows the principle of controlled, bounded exploration, prioritizing clear and interpretable insights over exhaustively searching an unbounded policy space. This approach is chosen for efficiency and for its focus on revealing fundamental trade-offs in system behavior.

The process is designed with the following considerations:

1. **Factor Definition:** Identify key policy factors that represent important, controllable operational trade-offs. Factors are grouped into distinct categories, such as those controlling resource supply (e.g., host topology, overcommitment levels) versus those governing placement logic (e.g., consolidation vs. load-spreading preferences). For each factor, select a limited number of representative levels to keep the experimental space compact and high-signal, creating strong points of comparison instead of a noisy continuous sweep.

2. **Configuration Generation:** Combine the chosen factors and levels into a bounded factorial set of experimental configurations. This enables systematic isolation of each policy's effect: by comparing configurations where only one factor changes at a time, observed differences in performance or energy can be attributed to specific design decisions. This greatly enhances interpretability and yields more robust, defensible conclusions.

3. **Simulation and Ranking:** Execute the full configuration set in the simulator under deterministic conditions (e.g., fixed random seeds and execution order) to control variance and ensure fair policy comparisons. After execution, apply a ranking module with pre-declared performance and energy criteria (per FR6). The aim is to identify a small set of Top-$k$ candidates that lie on or near the Pareto frontier of performance vs. energy efficiency, rather than merely sorting by a single metric. These candidates form the input to the subsequent physical output fidelity validation phase.

This structured exploration provides the necessary input for output fidelity validation, which ultimately determines the framework's decision-support value. The specific experimental factors and levels are detailed in the Evaluation chapter.

### 3.6.2 Design for Output Fidelity Validation

The second and final phase of the Evaluation stage is the design of a resource-efficient output fidelity validation process. This phase is the crucial step that "closes the loop," designed to empirically verify that the insights and rankings generated by the simulator hold true in the physical world. This provides the ultimate evidence for the methodology's decision-support value.

To ensure efficiency, the design specifies a small-sample output fidelity validation methodology, where only the Top-k candidates identified in the design space exploration phase are deployed and executed, rather than re-running the entire factorial set of configurations. The primary goal of the assessment in this phase is not to re-evaluate the simulator's absolute accuracy. Instead, the core evaluation criterion is ranking consistency. The assessment is designed to verify whether the relative performance and energy trade-off ranking of the Top-k candidates observed on the execution platform matches the ranking that was predicted by the simulation.

A consistent ranking serves as the final, evidence-backed confirmation that the proposed workflow can reliably guide operators towards superior system configurations. This confirmation fulfills the final functional requirement (FR7) and provides a quantitative answer to our research question regarding the methodology's practical utility. The detailed setup and results of this output fidelity validation are presented in the Evaluation chapter.

## 3.7 Summary of Design

This chapter has laid out the complete architectural blueprint for the trace-driven, closed-loop evaluation methodology. The design was guided by the core principles of modularity, semantic fidelity, and reproducibility, ensuring the resulting system is both robust and trustworthy.

The architecture is composed of several key conceptual components. At its heart is the Core Trace Schema (CTS), which was formalized as the standardized data interface between the physical and simulated worlds. A multi-stage data processing pipeline was designed to reliably transform raw, real-world data into this CTS format. The core of the framework's self-correcting capability lies in the input fidelity validation and feedback loop,

which provides a systematic process for quantitatively assessing the simulator's fidelity and applying controlled calibration. Finally, the chapter outlined a two-stage simulation-driven evaluation strategy for leveraging the validated simulator to perform efficient design space exploration and targeted output fidelity validation.

With this architectural blueprint now fully established, the following chapter will detail its concrete implementation, focusing on the specific engineering decisions, tools, and algorithms used to bring this design to life.

# 4

# Implementation

This chapter details the concrete implementation of the conceptual design presented in Chapter 3. It will describe the specific tools, algorithms, and data structures used to realize our CTS and processing framework, focusing on the practical "how" of our work to ensure transparency and reproducibility.

## 4.1 Technology Stack and Chapter Structure

Our implementation is primarily developed in Python (version 3.8 or newer), leveraging its rich libraries for scientific computing and data analysis. As established in Chapter 2, the system uses two core platforms: the physical execution platform is Continuum (version 1.0), and the simulator is OpenDC (version 2.4f). To ensure full reproducibility, a complete list of all software dependencies and their exact versions is provided in the accompanying code repository.

The following sections elaborate on the implementation of the architecture's core components. The discussion is structured to highlight the key engineering decisions and trade-offs made during the process of translating the conceptual design into a practical system. We will begin with the implementation of the Core Trace Schema (CTS), focusing on the rationale behind its data formats and invariants. Subsequently, we will describe the Data Processing Pipeline, detailing the strategies chosen for non-intrusive data collection and robust transformation. Finally, we will cover the Input Fidelity Validation and Feedback Loop Loop, explaining the motivations for selecting specific statistical metrics and calibration techniques.

## 4.2 Implementing the Core Trace Schema (CTS)

The implementation of the conceptual CTS design required translating its abstract components into concrete artifacts. This process was guided by two key engineering decisions: first, the selection of an appropriate data serialization format, and second, the precise specification of the data fields and integrity invariants.

### 4.2.1 Selecting the Data Serialization Format

A foundational challenge in implementing the CTS was choosing a data format that was both efficient for our processing pipeline and straightforward for researchers to inspect. To meet these dual requirements, we chose JSON (JavaScript Object Notation) and JSON Lines.

This choice was motivated by several factors. As text-based formats, they are human-readable, which is invaluable for manual debugging of trace data. They are also supported by a vast ecosystem of libraries in Python, simplifying the implementation of our processing scripts. The JSON Lines format, in particular, is highly advantageous for handling large trace files, as it allows for stream processing without needing to load the entire dataset into memory.

Based on this decision, we materialized the three logical components of the CTS into the following text-based artifacts:
`invocations.jsonl`: A JSON Lines file for task lifecycle events.
`proc_metrics.jsonl`: A JSON Lines file for resource usage samples.
`nodes.json`: A standard JSON file for the static environment description.

### 4.2.2 Defining the Schema Fields and Invariants

Beyond the format, a robust implementation required defining a concrete set of fields and, crucially, a set of rules to guarantee the logical consistency of the collected data. An inconsistent trace would invalidate any subsequent simulation results. Therefore, our next decision was to define a minimal but complete set of mandatory fields and a strict set of semantic and temporal invariants.

A minimal field set reduces data collection overhead and simplifies processing logic. The invariants, in turn, act as a formal "contract" for data quality. By enforcing these rules programmatically, we can automatically verify traces, ensuring that critical properties like temporal ordering (e.g., a task cannot end before it starts) and referential integrity are always preserved. This is a prerequisite for reproducible simulation.

The following tables specify the core fields defined for each artifact.

`invocations.jsonl` Each record must contain the following core fields to describe a task's lifecycle:

| Field | Type | Description |
|---|---|---|
| `trace_id` | string | A unique identifier for the invocation. |
| `pid` | integer | The operating system process identifier, used to link this record to the corresponding resource usage samples. |
| `ts_enqueue` | integer | The Unix timestamp in milliseconds when the task was submitted to a queue. |
| `ts_start` | integer | The timestamp in milliseconds when the task began execution. |
| `ts_end` | integer | The timestamp in milliseconds when the task finished execution. |

`proc_metrics.jsonl` Each record represents a single resource usage sample over a time window and must contain:

| Field | Type | Description |
|---|---|---|
| `ts_ms` | integer | The timestamp in milliseconds marking the end of the sampling window. |
| `pid` | integer | The process identifier to link the sample to a task. |
| `dt_ms` | integer | The duration of the sampling window in milliseconds, relative to the previous sample for the same `pid`. |
| `cpu_ms` | integer | The total CPU time (user + kernel) in milliseconds consumed by the process within the window. |
| `rss_kb` | integer | The Resident Set Size in kilobytes at the moment of sampling. |

`nodes.json` Each node description must contain:

| Field | Type | Description |
|---|---|---|
| `node_id` | string | A unique identifier for the node. |
| `cpu_cores` | integer | The total number of logical CPU cores. |
| `mem_mb` | integer | The total physical memory in megabytes. |
| `cpu_freq_mhz` | integer | The maximum CPU frequency in MHz. |

To enforce the data quality contract, all CTS artifacts must adhere to the following invariants:

**Timestamp Consistency:** All timestamps are integer Unix timestamps in milliseconds and should be recorded in a unified time zone (UTC is recommended).

**Lifecycle Order:** For every task, the timestamps must satisfy the logical order: `ts_enqueue` $\leq$ `ts_start` $\leq$ `ts_end`.

**Time-Series Monotonicity:** For the sequence of `proc_metrics` corresponding to a single `pid`, the `ts_ms` field must be strictly increasing.

**Referential Integrity:** Every `pid` in `invocations.jsonl` must have a corresponding set of samples in `proc_metrics.jsonl`, and vice-versa. Furthermore, the `ts_start` and `ts_end` of a task must fall within the time range of its corresponding samples.

Adherence to these invariants is automatically checked by our Quality Assurance module, as detailed in Section 4.3.2. Examples of the resulting CTS artifacts are provided below.

`invocations.jsonl` and `proc_metrics.jsonl` use JSON Lines; `nodes.json` uses JSON (object or array). All timestamps are integer milliseconds. Examples (non-essential fields omitted for readability):

```
{"trace_id":"job-0001","pid":32104,"ts_enqueue":1726128005000,"
   ts_start":1726128007100,"ts_end":1726128009320}
```

**Listing 4.1:** `invocations.jsonl`

```
{"ts_ms":1726128007100,"pid":32104,"dt_ms":200,"cpu_ms":180,"rss_kb
   ":420000}
{"ts_ms":1726128007300,"pid":32104,"dt_ms":300,"cpu_ms":220,"rss_kb
   ":421200}
```

**Listing 4.2:** `proc_metrics.jsonl` (adjacent samples for the same `pid`)

```
[
  {"node_id":"n-1","cpu_cores":8,"mem_mb":16384,"cpu_freq_mhz
     ":3500},
]
```

**Listing 4.3:** `nodes.json`

## 4.3 Implementing the Data Processing Pipeline

Implementing the conceptual design from Chapter 3 required a series of key engineering decisions to handle the practical challenges of data collection, standardization, and adap-

tation. This section details these decisions and their resulting implementations, starting with the crucial first step: data collection.

### 4.3.1 Design and Implementation of the Data Collection Strategy

The first implementation challenge was to reliably capture a complete, time-aligned picture of system behavior from the physical experiment running on Continuum. This led to our first key design decision: to adopt a multi-layered, non-intrusive data collection strategy. We implemented a set of concurrently running scripts to capture data from three distinct logical sources: the application, the operating system, and the hardware/virtualization layer.

**Application-Layer Data Collection** To capture high-precision task lifecycle events without modifying the application's source code, we decided to implement a non-intrusive Python wrapper script. Our current implementation, `tools/adapters/ffmpeg_wrapper.py`, serves as a proof-of-concept tailored for batch-processing applications like FFmpeg. The operational logic is as follows :

1. **Initiation:** The wrapper is invoked by a scheduler component (`worker.py`), which injects the submission timestamp into the `TS_ENQUEUE` environment variable. The wrapper script reads this value to record `ts_enqueue` .

2. **Process Spawning:** The wrapper uses Python's `subprocess.Popen` to launch the actual application as a child process, immediately creating a new Process Identifier (PID).

3. **PID Sentinel Creation:** Upon launch, the wrapper creates a "sentinel file" named after the PID in a shared directory (`logs/$RUN_ID/pids/<the_pid>`). This mechanism serves as a low-overhead signal for our system collector, which will be detailed next .

4. **Start Timestamp (`ts_start`) Calculation:** To get a precise start time, the wrapper reads the `starttime` field from `/proc/<the_pid>/stat` and combines it with the system's boot time and clock tick rate to calculate the absolute Unix timestamp.

5. **Termination and End Timestamp (`ts_end`):** The wrapper script calls the `.wait()` method on the child process, pausing until the application terminates. Upon waking, it immediately records the current wall-clock time as `ts_end` .

6. **Event Logging:** Finally, the wrapper appends a single JSON Lines record with all collected lifecycle data to a log file.

**Operating System-Layer Data Collection** A key challenge at this layer was to monitor process-level metrics efficiently without the high overhead of a system-wide scan. To solve this, we designed a targeted sampling mechanism based on the "sentinel files" created by the application wrapper. This was implemented in a separate, long-running Python script (`tools/collect_sys.py`) which acts as a background daemon with the following logic:

- **Targeted Sampling:** The script is configured to monitor the PID sentinel directory. This "whitelist" approach ensures monitoring resources are focused exclusively on the processes relevant to the experiment.
- **Periodic Collection:** In a continuous loop, the script wakes at a configurable interval (e.g., 1000ms). For each active PID in its whitelist, it reads and parses the relevant `/proc` files to obtain metrics like cumulative CPU time and RSS .
- **Data Logging:** At each sampling moment, the collector appends a separate JSON Lines record for each monitored PID to the `proc_metrics.jsonl` log file.

**Hardware-Layer Data Collection** Because virtual machines are abstracted from the physical hardware, they lack direct access to the host's power measurement units, such as Intel's RAPL counters. This led to the architectural decision that accurate energy accounting must be performed at the host level . Our implementation achieves this by combining Scaphandre with a lightweight local sampler script (`vm_energy_tools.py`). Scaphandre is run on the host in its QEMU integration mode, where it continuously reads the Intel RAPL energy counters (`energy_uj`) and attributes the host's total consumption to individual `qemu-kvm` processes . Our custom sampler script then polls the files exposed by Scaphandre at a fixed interval, calculates the energy delta since the last sample, and divides by the elapsed time to compute the average power in Watts, writing the final output to a CSV file .

### 4.3.2 Post-Processing: Standardization and Quality Assurance

With the raw log files collected, we faced the next core engineering challenge: how to transform the separate, and potentially inconsistent, log streams into the standardized

and trustworthy CTS artifacts required for simulation. To address this systematically, our first design decision was to implement a dedicated post-processing stage, orchestrated by a central Python script, `tools/parse_sys.py`.

A second, more critical architectural decision governed the design of this stage: the separation of standardization from adaptation. We chose to have this stage produce CTS artifacts that are a normalized, general-purpose representation of the experiment. More complex, simulator-specific tasks, such as joining task lifecycles with their time-windowed resource samples, are intentionally deferred to the next adaptation stage. This decision ensures the reusability and modularity of the CTS artifacts; once generated, they can serve as a platform-agnostic "golden source" to support different simulators or analysis tools in the future. Finally, to ensure the process is transparent and reproducible, this stage is also responsible for generating a crucial audit report.

**CTS Artifact Generation** The script's primary role is to generate the three clean CTS artifacts from the raw inputs, with each artifact requiring a distinct processing logic:
`invocations.jsonl`: To create a unified and standardized record of all task lifecycles, the script first consolidates the `events.*.jsonl` files produced by the distributed application wrappers. A key step in this process is a field-subsetting operation, which retains only the five core fields required by the CTS (`trace_id`, `pid`, `ts_enqueue`, `ts_start`, and `ts_end`). This not only enforces the schema but also simplifies downstream processing by filtering out extraneous application-specific data.

`proc_metrics.jsonl`: Generating this artifact involved the most critical transformation. Raw OS metrics, such as CPU time, are reported as monotonically increasing, cumulative counters. This format is not directly usable for simulation, which requires discrete usage fragments for each time window. To bridge this semantic gap, we implemented a differencing algorithm. For each process (`pid`), the algorithm iterates through its samples chronologically, calculating the delta between consecutive measurements. This yields the per-window `dt_ms` (time delta) and `cpu_ms` (CPU time delta, converted using the system's `CLK_TCK`), making the raw data suitable for a faithful simulation replay.

`nodes.json`: Finally, to ensure the simulation baseline is equivalent to the physical environment, the `nodes.json` artifact is generated by querying system utilities once per run for their static properties. This design decision cleanly decouples the dynamic workload trace from the static infrastructure description, which is a crucial principle for conducting reproducible "what-if" experiments on different simulated topologies.

**Quality Assurance Implementation** As the final step of the post-processing stage, the generated CTS artifacts are subjected to an automated Quality Assurance (QA) module. The engineering decision to make this a mandatory, programmatic check—rather than relying on manual inspection—is crucial for guaranteeing the trustworthiness of our entire framework. This module acts as a gatekeeper, programmatically enforcing the data quality contract defined in Section 4.2.2 and preventing corrupted or logically inconsistent data from propagating to the simulation stage.

The validation process is multi-faceted. First, it verifies field completeness, ensuring all required fields are present in every record. Next, it enforces temporal consistency, checking for logical impossibilities such as a task ending before it starts (`ts_enqueue` < `ts_start` < `ts_end`) and ensuring all time-series data are monotonic. Finally, it validates cross-reference integrity, identifying any "orphaned" `pid` in one artifact that cannot be matched in the other, which would cause errors during simulation replay. The results of all checks are compiled into a Markdown audit report, making the transformation process fully transparent and reproducible.

### 4.3.3 Implementing CTS Adaptation for the Simulator

With the standardized CTS artifacts prepared, the final phase of the data processing pipeline is to adapt them for the target simulator, OpenDC. This adaptation is not merely a format conversion but a crucial architectural decision. By implementing it as a distinct, final stage, we intentionally decouple our general-purpose CTS from the proprietary requirements of any single simulator. This design choice preserves the modularity of our framework, ensuring that the CTS artifacts can serve as a reusable, platform-agnostic source for other analysis tools or simulators in the future.

This "translator" role is fulfilled by a dedicated Python script, `tools/export_opendp.py`. Its responsibility is to consume the three CTS artifacts and generate the specific Parquet-based trace files that OpenDC requires. The following subsections will first detail the target OpenDC format and then explain the transformation logic implemented to produce each file.

**Target Workload Format: OpenDC Parquet Traces** The implementation logic of our translator script is fundamentally shaped by the specific, proprietary format required by OpenDC's trace-driven workload model. Before detailing the transformation, it is essential to first describe this target format. The OpenDC model defines a simulation scenario using a set of three distinct files, each serving a unique conceptual purpose:

- `tasks.parquet`: This file defines the static metadata for each task. Each row represents a single task and specifies its high-level properties, such as its unique `id`, `submission_time`, total `duration`, and its requested resource capacities (`cpu_count`, `cpu_capacity`, `mem_capacity`).
- `fragments.parquet`: This is the core component for modeling dynamic behavior. It breaks down each task's execution into a sequence of smaller "fragments." Each row represents one such fragment, defined by its `duration` (in ms) and a constant `cpu_usage` (in MHz), thereby describing the task's time-varying CPU demand.
- *Topology File*: This JSON file describes the static environment—the datacenter infrastructure where the workload will be replayed.

Consequently, the central engineering task of our script is to translate the three standardized CTS artifacts into these three specialized OpenDC files, a process detailed in the following sections.

**Generating `tasks.parquet`** The `tasks.parquet` file is generated by translating the event-based CTS artifacts into the high-level, static metadata that OpenDC expects. While some fields can be mapped directly, the key engineering challenge lies in deriving a single, representative resource capacity value for each task from its time-series usage data. The implementation for each field is as follows:

- **`id`, `submission_time`, `duration`**: These core lifecycle fields are derived directly from `invocations.jsonl`. `submission_time` is mapped from `ts_enqueue`, and `duration` is calculated as (`ts_end` − `ts_start`). The task `id` defaults to the original `pid`, with a configurable fallback to a unique sequential numbering scheme to handle potential PID reuse by the operating system.
- `mem_capacity`: To ensure simulation robustness (i.e., avoiding out-of-memory failures), we adopt a conservative peak-within-window strategy. For each task, the script finds the maximum Resident Set Size (`rss_kb`) recorded in `proc_metrics.jsonl` during the task's execution window, ensuring the simulator allocates sufficient memory.
- **`cpu_capacity` and `cpu_count`**: A simple mean of CPU usage would understate peak demand, while using the absolute maximum would be overly sensitive to transient spikes. To balance realism and robustness, we implement a two-stage statistical modeling strategy. After the time-series of CPU usage fragments is generated (see

the next section), the script computes the 95th percentile (P95) of `cpu_usage` across all fragments for each task and sets this value as `cpu_capacity` (total MHz across all cores). The `cpu_count` is then derived from this `cpu_capacity` and the node's known CPU frequency.

**Generating `fragments.parquet`** While `tasks.parquet` defines what tasks exist, `fragments.parquet` describes their dynamic behavior over time, which forms the core of the trace-driven simulation. The primary translation logic involves converting the time-series data from `proc_metrics.jsonl` into a sequence of OpenDC fragments. Our initial design decision was a direct sample-to-fragment mapping: for each valid sample within a task's execution window, a corresponding fragment is generated. The `cpu_usage` for the fragment is calculated based on the CPU time consumed during the sampling interval (`cpu_ms`/`dt_ms`) and then converted to MHz.

However, a robust implementation must account for the inherent limitations of discrete sampling. A simple one-to-one mapping is insufficient because it can lead to data loss and an incomplete workload. We therefore implemented two additional strategies to handle critical edge cases:

- **Head Fragment Padding:** There is often a delay between a task's actual start time (`ts_start`) and the collection of its first resource sample. To prevent this initial, unmeasured period of execution from being ignored, our script synthesizes a "head" fragment to fill this gap. As a reasonable heuristic, the script assumes the initial CPU usage is equivalent to the first measured usage, and thus populates this head fragment with the `cpu_usage` value from the immediately following fragment.

- **Synthetic Fragment Generation:** A more critical edge case involves very short-lived tasks that may start and finish between two consecutive sampling ticks, resulting in no recorded samples. A naive mapping would drop these tasks from the simulation entirely, compromising its fidelity. To prevent this, our script makes a conservative assumption: it generates a single synthetic fragment spanning the task's known duration, with an assigned `cpu_usage` equivalent to half of the host's maximum CPU frequency.

These corrective measures are crucial engineering decisions that guarantee the completeness and fidelity of the replayed workload, ensuring that every task from the physical run is "sim-drivable" and faithfully represented in the simulation.

**Table 4.1:** `tasks.parquet`.

| id | submission_time | duration | cpu_count | cpu_capacity | mem_capacity |
|---|---|---|---|---|---|
| 145156 | 1758059356963 | 100959 | 1 | 2400 | 129984 |

**Table 4.2:** `fragments.parquet`.

| id | duration | cpu_usage |
|---|---|---|
| 145156 | 727 | 2379.9603174603176 |
| 145156 | 1008 | 2379.9603174603176 |
| 145156 | 1001 | 2420.5694305694306 |

**Generating the Topology File** Finally, a valid and reproducible comparison requires that the simulated infrastructure be a precise replica of the physical environment where the trace was collected. To enforce this equivalence and eliminate the risk of manual configuration errors, our script implements a key design choice: it uses the `nodes.json` artifact as the single source of truth to programmatically generate the simulator's topology file.

The process is fully automated: the script aggregates all node specifications from the file, grouping hosts with identical configurations (cores, frequency, memory) into a concise cluster definition with a `count` field. This approach guarantees that the simulated environment is a verifiable and precise match for its physical counterpart, which is a cornerstone for the trustworthy input fidelity validation presented in the following chapters.

### 4.3.4 Example Transformation Output

To make the abstract transformation process tangible, this section provides a concrete example of the final OpenDC-specific outputs. The following tables 4.1 and 4.2 show the content of the `tasks.parquet` and `fragments.parquet` records that are generated from the CTS artifacts. As the Parquet format is binary, the data is presented here in a tabular format for readability.

## 4.4 Implementation of the Input Fidelity Validation and Feedback Loop Loop

Having successfully translated our real-world traces into a simulator-ready format, the workflow's focus now shifts from data preparation to formal fidelity validation. The availability of a trace does not in itself guarantee a faithful simulation; a rigorous, empirical

process is required to ensure the simulation model is a trustworthy representation of its physical counterpart. This section, therefore, details the concrete implementation of this crucial Input Fidelity Validation and Feedback Loop Loop, as designed in Section 3.5.

We will describe the implementation of the "Input Fidelity Validation Engine"—a collection of components working in concert to systematically bridge the physical and simulated worlds. The following subsections detail the end-to-end workflow of this engine: from replaying the trace in OpenDC to generate simulation data, to calculating aligned metrics, performing automated consistency assessments, and finally, applying a systematic model calibration when needed.

### 4.4.1 Implementing the Trace Replay in OpenDC

The successful generation of the `tasks.parquet`, `fragments.parquet`, and topology JSON files in the previous stage provides all the necessary inputs to drive a simulation. The primary goal of this stage is to execute a simulation run under conditions that are precisely controlled to mirror the physical experiment. This section describes the implementation of the trace replay process in OpenDC, which serves to generate the raw simulation data required for the input fidelity validation against the physical experiment.

The trace replay is orchestrated using OpenDC's batch execution module. A complete simulation run requires the following input artifacts:

- **Workload Files:** The `tasks.parquet` and `fragments.parquet` files, generated as described in Section 4.3.3, define the set of tasks and their time-varying resource demands.

- **Topology File:** A JSON file, also generated from the `nodes.json` artifact, which describes the simulated datacenter's physical infrastructure. For the input fidelity validation, this topology is configured to be an exact replica of the Continuum environment, matching the number of hosts, as well as their CPU core counts, frequencies, and memory capacities.

- **Experiment Definition File:** A central JSON file that defines the simulation scenario. It specifies which workload and topology to use, the duration of the simulation, and, crucially, the policies to be applied, such as the task scheduling algorithm. To establish a valid baseline for comparison, the policies configured in this file are chosen to mirror the behavior of the physical execution platform as closely as possible.

The detailed exploration of different policy configurations will be presented in the Evaluation chapter.

The simulation is executed using the `OpenDCExperimentRunner`, a native OpenDC command-line tool that automates the process of running experiments defined in the aforementioned file. A typical invocation is managed by a script as follows:

```
# Example command to run the experiment
OpenDCExperimentRunner.bat --experiment-path experiments/baseline_validation.json
```

Upon completion, the runner generates a series of detailed time-series data files. These files are not merely logs; they represent the raw evidentiary basis for the entire validation process. For the specific validation goals of our methodology, our "Input Fidelity Validation Engine" is engineered to consume two primary outputs:

- **Task Metrics:** Provides detailed lifecycle data (submission, start, finish times) and fine-grained resource usage for each simulated task. This serves as the direct source for key fidelity checks: macroscopic performance metrics (task wait time distribution, cumulative completion curve) are derived from the lifecycle events, while the microscopic validation of instantaneous CPU usage is based on the usage data.
- **Power Source Metrics:** Contains a time-series record of the total power consumed by datacenter components. This data provides the essential basis for validating the dynamic behavior and trend accuracy of the simulator's energy model against the physical measurements.

### 4.4.2 Implementation of Aligned Metrics Calculation

The raw outputs from the physical and simulated runs provide the necessary data, but not yet the insight. A direct comparison of raw log files is infeasible; a meaningful, quantitative validation requires transforming this data into the set of semantically aligned metrics defined in our design (Section 3.5.1) .

This crucial processing step is implemented as a dedicated metrics calculation layer within our Input Fidelity Validation Engine. We made the engineering decision to build this layer using Python, leveraging the powerful data manipulation capabilities of the Pandas library, to create a robust and repeatable analysis workflow. The following subsections detail the specific logic used to calculate each of the four key fidelity metrics from their respective data sources.

**Task Wait Time**

The first key metric targets the simulator's ability to faithfully reproduce the queuing dynamics of the physical system. We assess this by comparing the statistical distribution of Task Wait Time, which is conceptually defined as the duration between a task's submission and the start of its execution.

- **For the execution platform (Continuum):** The wait time for each task is computed by subtracting the `ts_enqueue` field from the `ts_start` field, sourced from the `invocations_merged.jsonl` file.
- **For the simulator (OpenDC):** The calculation uses the `task.parquet` output file. The wait time is the difference between the `schedule_time` and `submission_time` for each task that reached a "COMPLETED" state.

This process results in two distinct data distributions, which serve as the direct inputs for the quantitative consistency assessment detailed in Section 4.4.3.

**Cumulative Completion Curve and Throughput**

Moving from individual task queuing to a macroscopic view, the next metric evaluates the simulator's fidelity in capturing the overall system throughput and final makespan. The primary tool for this is the Cumulative Completion Curve, which tracks the total number of completed tasks over time.

The implementation of this metric involves a key design choice: rather than plotting a simple step-function of individual completion events, which can be noisy and difficult to compare, we adopted a time-binning and aggregation strategy to produce smooth, comparable time series. This process is implemented in three logical steps:

1. First, the completion timestamps of all tasks are extracted, carefully mapping Continuum's `ts_end` field and OpenDC's `finish_time` field to ensure semantic equivalence.
2. Next, these timestamps are binned into fixed-duration intervals (e.g., 60 seconds) to calculate the throughput—the number of tasks completed—within each time window.
3. Finally, a cumulative sum is computed over these throughput bins to construct the completion curve for each platform.

This method effectively transforms discrete completion events into two directly comparable time-series representations of system performance.

**Instantaneous CPU Usage**

Having validated the simulation's macroscopic outcomes, we now turn to a microscopic

validation of the simulator's execution fidelity. This metric is crucial because it directly assesses whether the simulator, when driven by our trace, behaves in a way that faithfully reproduces the fine-grained resource consumption of the real system.

To achieve this, we compare the time-series of CPU usage as reported by both the physical and simulated environments:

- **For the execution platform (Continuum):** The CPU usage is derived from the `proc_metrics_merged.jsonl` file. This file is pre-processed to include the host's `cpu_freq_mhz` for each entry. The instantaneous usage is then calculated using the formula: $\texttt{cpu\_usage\_mhz} = (\texttt{cpu\_ms}/\texttt{dt\_ms}) \times \texttt{cpu\_freq\_mhz}$.
- **For the simulator (OpenDC):** The CPU demand is directly available in the `cpu_usage` column of the `task.parquet` output file.

A primary challenge in comparing these two time-series is achieving a fair and meaningful alignment. A comparison on a global, absolute timeline would be flawed, as minor, legitimate scheduling delays could cause a persistent mismatch. To overcome this, our implementation adopted the per-task, relative time alignment strategy designed in Chapter 3.

The implementation is realized through a data transformation script. For each unique task (identified by its `pid`), the script first establishes an independent, relative time axis by calculating the elapsed time since that specific task's execution start (`ts_start`). The physical and simulated CPU usage data points for that task are then re-indexed according to this relative time. Finally, data points from both sources that share the same relative timestamp (at one-second granularity) are paired together.

This process yields a collection of correctly aligned data pairs for each task, forming a robust basis for the quantitative consistency assessment in the next section.

**Total Power Draw Trend**

The final metric assesses the fidelity of the simulator's energy model. A direct comparison of absolute power values in Watts is often brittle and not generalizable, as these measurements are highly dependent on the specific physical hardware. For a more robust and hardware-agnostic validation, our methodology therefore focuses on comparing the dynamic trend and shape of the power curves, rather than their absolute magnitudes. This is achieved by normalizing both time-series using a z-score transformation, which allows us to evaluate their correlation regardless of their scale.

However, the raw physical power data required pre-processing before it could be normalized. The data, sourced from Intel RAPL's cumulative energy counter (`energy_uj`), does

not update every second, resulting in an unrealistic signal with many zero-power readings followed by intermittent spikes. To convert this into a more realistic, continuous signal, we implemented a data reconstruction algorithm. This algorithm calculates the average power between two consecutive counter updates and back-fills this value for every second within that interval. The total system power is then the sum from all nodes. In parallel, the simulator's power data, read from the `powerSource.parquet` file, is aggregated by the second to produce a time series of equivalent granularity.

After this reconstruction of the physical data and aggregation of the simulated data, both time-series are normalized. This process yields two clean, comparable Z-scored signals that serve as the direct input for the segmented consistency assessment detailed in Section 4.4.3.

### 4.4.3 Implementation of Consistency Assessment Criteria

With the aligned metrics calculated, the Input Fidelity Validation Engine is now ready to perform its most critical function: rendering a formal judgment on the simulator's fidelity. This is accomplished through an automated consistency assessment, which acts as the decision point of the entire input fidelity validation loop.

To avoid subjective interpretation of results, we established a set of pre-declared, quantifiable criteria for each metric. These criteria serve as a formal contract defining what constitutes an acceptable level of consistency. Our analysis scripts systematically evaluate the metrics against these thresholds, producing a clear, binary PASS/FAIL judgment.

This outcome is not merely informational; it programmatically directs the workflow. A PASS across all metrics signifies that the simulator is a faithful representation that has passed input fidelity validation, allowing the framework to proceed to the Evaluation stage (Chapter 5). A FAIL on any metric, however, triggers the systematic parameter calibration loop detailed in Section 4.4.4. The results of all checks are compiled into a Markdown audit report, ensuring the final judgment is both transparent and reproducible

**Task Wait Time Distribution**

The evaluation of the task wait time distributions is implemented using a dual-criterion approach. This design choice reflects the need for a comprehensive assessment, as relying on a single statistical measure can be misleading. A simple comparison of averages, for instance, could hide significant differences in the shape of the distributions, while a shape-only test might not capture deviations at critical points like the tail. Our approach therefore combines two complementary forms of validation:

- **Distributional Shape:** To assess the macroscopic similarity of the distributions, a two-sample Kolmogorov–Smirnov (KS) test is performed. The distributions are considered consistent if the resulting p-value is greater than or equal to 0.05.

- **Quantile Error:** To ensure fidelity at specific, critical points—especially the median (p50) and the long tail (p95, p99), which heavily impacts user experience, the relative error of the 50th, 95th, and 99th percentiles (p50, p95, p99) of the simulated distribution is calculated against the physical distribution. The errors must remain within the thresholds of 5%, 10%, and 20%, respectively.

A PASS for this metric is granted only when both of these conditions are met, ensuring that the simulation's queuing behavior is a robust match to the execution platform.

**Cumulative Completion Curve and Throughput**

To assess the alignment of macroscopic performance, we implemented a two-part validation that evaluates both the progression of work over time and the final completion point. This ensures the simulation not only finishes at the right time but also follows the correct throughput trajectory.

- **Curve Similarity:** The primary criterion evaluates the similarity of the two cumulative completion curves. The Root Mean Square Error (RMSE) between the two cumulative completion curves is calculated. To account for minor, consistent system latencies, the script first finds the optimal integer time-bin lag that minimizes this RMSE. The final metric, RMSE%, is the best-lag RMSE normalized by the total number of tasks completed on the execution platform. A PASS requires this value to be no more than 3%.

- **Makespan Accuracy:** While curve similarity ensures the trend is correct, a second criterion validates the absolute endpoint. The total engineering makespan is evaluated using a dual-threshold rule: for short workloads ($< 10$ min), the absolute error must not exceed 6 seconds; for longer workloads, the relative error must be within 1%.

A successful validation for this metric requires both criteria to be satisfied.

**Instantaneous CPU Usage**

The assessment for instantaneous CPU usage is implemented as a stringent, multi-faceted validation, applied to the set of per-task, relatively-aligned data pairs generated in the previous section. This alignment strategy provides a much fairer basis for comparison,

ensuring that we are evaluating the fidelity of the task's intrinsic behavioral model, rather than artifacts of scheduling jitter.

Our implementation remains rooted in the understanding that a single error metric can be misleading. To ensure a truly robust validation, our criteria test different, complementary aspects of fidelity:

- **Trend Consistency (Pearson Correlation):** To confirm that the simulated usage correctly follows the dynamic trends of the physical usage (i.e., they rise and fall together), we calculate the Pearson correlation coefficient $r$. A strong positive linear relationship, indicated by a coefficient of $r \geq 0.95$, is required.

- **Point-wise Accuracy (SMAPE):** To measure the average magnitude of error on a point-by-point basis, we use the Symmetric Mean Absolute Percentage Error (SMAPE). This metric provides a clear indication of the typical percentage deviation. The assessment passes if SMAPE $\leq 5\%$.

- **Error Magnitude (Relative RMSE):** As a complementary measure of error that is more sensitive to large deviations, we calculate the Root Mean Square Error (RMSE), normalized by the median physical usage. This value must not exceed 10%.

A PASS for this metric is granted only when all three criteria are met simultaneously. This strict requirement, combined with our robust alignment methodology, ensures that the simulation faithfully represents the task's behavior not only in its overall trend but also in its point-by-point accuracy.

**Total Power Draw Trend**

The implementation for assessing the total power draw trend required a more sophisticated approach than a simple global correlation. A single correlation coefficient, even when optimized for lag, could conceal significant periods of poor alignment, failing to guarantee the sustained fidelity that our design mandates. To address this, we implemented the *Segmented Correlation Assessment* methodology as a multi-stage automated script.

The script's first engineering decision is to partition the entire time-series into fixed-duration, non-overlapping segments (e.g., 300 seconds). This was crucial for enabling a more granular analysis, allowing us to independently verify the model's performance during distinct phases of the workload, such as task bursts versus idle periods.

The next challenge was to handle minor temporal misalignments robustly. Rather than applying a single, brittle time-shift across the entire dataset, the script performs a *per-segment best-lag search*. This was a critical implementation choice to make the assessment resilient to non-constant delays or minor clock drift between the physical and simulated systems—artifacts that a global lag would fail to correct. For each segment, the script searches within a predefined window (e.g., ±600 seconds) to find the time offset that maximizes the Pearson correlation, thereby identifying the true local trend alignment.

Finally, to produce a single, decisive metric for judgment, the script aggregates the best-lag correlation coefficient $r_k$ from each valid segment. We chose the mean, mean($r_k$), as the primary output for the final assessment. This decision reflects the design goal of ensuring consistent performance; the mean of segmental scores provides a much stronger guarantee of fidelity throughout the experiment than a single, global value might. Based on this, a **PASS** is granted if this mean value meets or exceeds the pre-declared threshold of 0.90.

### 4.4.4 Implementation of the Parameter Calibration Method

Should the automated consistency assessment from the previous section return a "FAIL" result, it indicates that the initial discrepancy between the physical and simulated systems exceeds the acceptable thresholds. In this event, the architecture's closed-loop design requires a systematic process to improve the simulator's fidelity. This section details the implementation of this corrective feedback mechanism: the parameter calibration method designed in Section 3.5.3. The guiding principle of this implementation is a **post-hoc, input-transformation-based** approach. Rather than modifying the OpenDC simulation kernel, the mechanism operates by applying a one-time, systematic transformation to the simulator's input artifacts for a given baseline run. This ensures modularity, transparency, and reproducibility.

The calibration is implemented as a suite of Python scripts, each targeting a specific input artifact and a corresponding model parameter.

**Tunable Parameters and Implementation**

A systematic calibration required us to first identify the most common sources of discrepancy between the physical and simulated systems. Our analysis revealed that these discrepancies typically fall into two distinct categories. Consequently, we designed and implemented a set of tunable parameters targeting both: those that adjust the simulation's behavior by transforming its inputs, and those that correct for systematic biases during the validation comparison itself.

1. **Simulation Input Transformation Parameters**

   The primary set of parameters modifies the simulation's input artifacts to correct for systemic differences in the model's behavior:

   - **System Speed Factor** ($a_{\text{speed}}$): To correct for macroscopic deviations in overall processing speed (i.e., makespan), this parameter scales the `coreSpeed` attribute of each host in the topology JSON file.
   - **Service Time Factor** ($a_{\text{service}}$): To correct for systemic differences in how long individual tasks take to execute, this parameter scales the *duration* of every workload fragment in the `fragments.parquet` file.
   - **Fixed Startup Overhead**: To account for unmodeled, constant delays at the beginning of a task's execution (e.g., OS process creation overhead), this parameter adds a fixed millisecond value to the duration of each task's first fragment.
   - **Effective Capacity Ratio** ($r_{\text{capacity}}$): To align the simulator's effective throughput with the physical system, this parameter modifies the resource capacity filters within the experiment's JSON definition file.

2. **Validation-Time Alignment Parameters**

   This second set of parameters is applied during the validation script's execution to align the data streams for a fairer comparison, without altering the simulation run itself:

   - **CPU Usage Scaling Factor** ($k$): To correct for linear scaling differences between the physically measured and simulated CPU usage values, this factor is applied to the physical CPU data before computing error metrics.
   - **Fixed Time Offset** ($\Delta t_{\text{ms}}$): To correct for constant time-base misalignments between the two time-series, this offset is applied to the timestamps of the physical data during the alignment step.

**Calibration Workflow**

The calibration process is implemented not as a fully automated black box, but as a systematic, operator-guided iterative workflow. This design pragmatically combines the system's systematic guidance with the operator's expertise to efficiently converge on a model that passes input fidelity validation when the initial consistency assessment returns a FAIL. The workflow proceeds in a tight loop of diagnosis, adjustment, and re-evaluation:

- **Diagnosis:** The process begins with diagnosis. The operator consults a pre-defined "decision recipe" which acts as a diagnostic guide, mapping specific failed metrics (the symptoms) to the most likely tunable parameters (the potential causes). For instance, a high RMSE in the cumulative completion curve points towards an adjustment of the System Speed Factor ($a_{\text{speed}}$).
- **Adjustment and Re-evaluation:** Based on the diagnosis, the operator applies a targeted adjustment by running the corresponding calibration script. The system then re-runs the entire input fidelity validation pipeline from simulation to consistency assessment—using the newly generated, calibrated inputs. This provides immediate, quantitative feedback on the efficacy of the adjustment.
- **Iteration:** This diagnostic loop is repeated in small increments until all consistency criteria from Section 4.4.3 are satisfied. Once the model is successfully calibrated, the parameter values are "frozen" for all subsequent evaluation experiments.

To ensure this entire workflow is transparent and reproducible, the implementation adheres to strict artifact management rules. Calibration scripts never overwrite original files; instead, they generate new files with a `_cal` suffix (e.g., `topology_cal.json`). Furthermore, each script execution produces a `manifest.json` file, an auditable record containing the exact parameter values used, ensuring that any calibrated experiment can be precisely reproduced.

### 4.4.5 Tools for Design Space Exploration

The implementation of the Design Space Exploration (DSE) stage addresses a fundamental shift in scale: from validating a single baseline scenario to systematically managing dozens of experimental runs and their corresponding outputs. To handle this complexity, our implementation is architecturally divided into a two-phase, automated workflow. The first phase focuses on the efficient generation of experimental data, while the second is dedicated to the automated analysis and distillation of insights from that data. This structure ensures a scalable and reproducible pipeline from experimental design to final, ranked conclusions.

**Experiment Definition and Execution**

A key advantage of OpenDC is its support for defining complex factorial experiments within a single experiment definition file. This feature obviates the need for custom scripts to generate separate configuration files for each experimental run. To explore the design space, we define all experimental "factors" and their "levels" directly within this central

JSON file, which automatically generates the Cartesian product of all combinations. Execution of the entire batch of experiments is then handled by a single invocation of the `OpenDCExperimentRunner` command, identical to the one used for the input fidelity validation in Section 4.4.1. This streamlined process demonstrates the efficiency of the workflow in launching large-scale simulation studies.

**Automated Results Analysis and Ranking**

After the batch simulation completes, a two-stage process, implemented by a pair of complementary Python scripts, is used to automatically parse, aggregate, and analyze the large volume of output data.

- **Results Aggregation (`analysis/compare_combined_results.py`):** This first script systematically scans the output directories of all experimental runs. It uses the Pandas library to read the key output files (e.g., `task.parquet`) and computes a set of macro-level KPIs for each run, such as P95 task wait time, total makespan, and energy consumed per task. These per-run metrics are then aggregated across different random seeds for each unique configuration, producing a final, clean summary table (`summary_by_config.csv`).
- **Higher-Level Analysis (`analysis/rank_configs.py`):** This second, more advanced script consumes the aggregated summary table to facilitate the identification of top-performing candidates. Rather than acting as a black-box ranker, it serves as a powerful analytical tool offering several methodologies to explore the design space:
  - **Pareto Optimal Filtering:** Programmatically isolates the set of non-dominated solutions. This output directly enables the visualization of the Pareto frontier, as presented in the evaluation in Section 5.3.1.
  - **Weighted Sum Scoring & Lexicographic Sorting:** Provides configurable, quantitative lenses to explore trade-offs under different priority schemes (e.g., "performance is twice as important as energy").

The final output of this stage is not a single ranked list, but rather a set of enriched analytical artifacts—including the set of Pareto-optimal points and the aggregated summary table itself—that serve as the direct, quantitative foundation for the final selection of candidates for output fidelity validation, as discussed in Chapter 5.

The final output of this stage is a ranked CSV file, presenting the Top-$k$ configurations that are candidates for output fidelity validation.

## 4.5 Summary of Implementation

This chapter detailed the concrete implementation of the closed-loop evaluation methodology, translating the conceptual design from Chapter 3 into a tangible and automated system. The focus was not merely on the implementation facts, but on the key engineering decisions and architectural principles that ensure the resulting system is robust, reproducible, and trustworthy.

The implementation was built upon a series of core design choices. A standardized Core Trace Schema (CTS) was established as the central data contract, with a robust data processing pipeline designed to bridge the semantic gap between raw, cumulative OS metrics and the discrete inputs required for simulation. This pipeline incorporates a crucial differencing algorithm and systematic handling of edge cases to ensure the fidelity of the final workload trace. To guarantee the simulator's trustworthiness, an automated Input Fidelity Validation Engine was implemented, which programmatically assesses the model's fidelity against pre-declared, objective criteria. This is complemented by a transparent and non-intrusive parameter calibration workflow, which enables a systematic, post-hoc correction of the model by transforming its inputs rather than modifying the simulator kernel. Finally, to support large-scale experimentation, a streamlined workflow for Design Space Exploration was implemented, leveraging a declarative approach for experiment definition and a dedicated analysis pipeline to process results and facilitate the identification of top-performing candidates.

With this fully implemented and systematically designed system now in place, the stage is set for a comprehensive experimental evaluation. The following chapter will use this entire system to quantitatively answer our core research questions regarding the methodology's accuracy and its ultimate decision-support value in a realistic cloud datacenter scenario.

# 5

# Evaluation

This chapter presents the comprehensive experimental evaluation of the framework designed and implemented in the preceding chapters. The primary goal of this evaluation is to quantitatively answer our third research question (RQ3): What is the impact of the proposed framework on the accuracy and decision-support value of performance and energy evaluations?

To address this question, we follow the two-stage evaluation strategy designed in Section 3.6. First, we conduct a rigorous input fidelity validation to establish the fidelity of our simulation model (Section 5.2). With this trust established, we then use the simulator to perform a wide-ranging Design Space Exploration (DSE) to identify optimal policy configurations (Section 5.3). Finally, we will select the top-ranked candidates from the DSE and perform output fidelity validation on them on the physical platform to assess the framework's end-to-end decision-support value (Section 5.4). This chapter begins by detailing the complete experimental setup that underpins all subsequent results.

## 5.1 Experimental Setup

This section details the complete setup used for our evaluation to ensure transparency and reproducibility. The following subsections describe the physical and simulation platforms upon which the experiments rely, the specific workload designed to test the system, and the core metrics used to measure the framework's fidelity and effectiveness.

### 5.1.1 Experimental Platforms

The core of this research is a closed-loop evaluation framework that bridges a physical environment with a digital simulation. Consequently, our experimental environment is

composed of a physical execution platform, used to generate the ground-truth baseline and for output fidelity validation, and a simulation platform, used for large-scale exploration.

**Physical Execution Platform: Continuum**

To generate high-fidelity and reproducible traces, we use the Continuum framework (version 1.0) as our physical execution platform. Continuum automates infrastructure deployment and benchmarking, providing an isolated and controllable environment for our experiments.

- **Core Functionality:** In this research, we use Continuum's VM-based abstraction to construct the required compute nodes. This approach ensures strong isolation between experimental runs and allows us to precisely control the resources (CPU, memory) and topology of each node via a declarative configuration file, thus guaranteeing reproducibility.
- **Hardware Configuration:** All physical experiments were conducted on a single server host with the following specifications:
  - **CPU:** Intel(R) Xeon(R) Silver 4210R @ 2.40GHz (40 Cores)
  - **Memory:** 256 GB DDR4
  - **Storage:** 447 GB SSD (System Disk) + 2 × 3.6 TB HDD (Experiment Data Disks)
  - **Host OS:** Ubuntu 22.04.1 LTS
- **Virtual Environment Configuration:**
  - **Virtualization Technology:** The experimental environment is deployed using QEMU/KVM virtual machines.
  - **Virtual Machine Specifications:** Unless otherwise specified, the baseline topology consists of two virtual machines, each allocated with 8 virtual CPU cores (at a frequency of 2400 MHz) and 32 GB of memory.
  - **Guest OS:** All virtual machines run Ubuntu 20.04.6 LTS to ensure a consistent software environment.
- **Energy Monitoring:** Host-level energy consumption data is collected using Scaphandre, which reads the processor's Intel RAPL (Running Average Power Limit) counters. This tool is capable of attributing the host's total energy consumption to the individual `qemu-kvm` processes managed by Continuum.

**Simulation Platform: OpenDC**

For efficient design space exploration, we employ OpenDC (version 2.4f) as our discrete-event simulation platform. OpenDC is designed for the modeling and simulation of cloud datacenters, with native support for the trace-driven replay of workloads and extensible models for resources and scheduling policies.

- **Environmental Equivalence:** To ensure a fair comparison between simulation and reality, the simulation environment is configured to be a precise "digital twin" of the physical execution platform.
- **Topology:** The simulated datacenter topology—including the number of hosts, CPU cores, core frequency, and memory capacity—is derived directly from the `nodes.json` environmental description file. This file is generated by capturing the static properties of the physical platform, guaranteeing a one-to-one correspondence of the infrastructure.
- **Workload:** The simulator is driven by `tasks.parquet` and `fragments.parquet` files. These files are produced by our data processing pipeline, which converts the raw traces collected on Continuum into the standardized Core Trace Schema (CTS) and then into the simulator-specific format.
- **Energy Model:** We utilize OpenDC's built-in energy model to estimate power consumption during the simulation. This model generates a power time-series based on CPU activity, which is used for trend comparison against the physical measurements.

### 5.1.2 Workload

With the physical and simulation platforms established, we now define the representative batch processing workload that is executed upon them to evaluate the framework's effectiveness. This workload simulates a common media processing scenario in cloud datacenters and consists of a series of FFmpeg video transcoding tasks.

**Source Video Generation**

For the sake of reproducibility and control, the source videos for our transcoding tasks were synthetically generated using a Python script (`tools/generate_test_videos.py`). This script uses FFmpeg's `lavfi` (libavfilter) virtual input, which does not require any external video files. Test videos with various visual patterns were generated based on a filtergraph. The base specifications for all generated source videos are as follows:

**Table 5.1:** Task types used in the workload.

| Task Type | Target Resolution | Target Codec | FFmpeg Preset | CPU Cores | Load Intensity |
|---|---|---|---|---|---|
| `light1c` | 480p | H.264 | veryfast | 1 | Lightest |
| `medium480p` | 480p | H.264 | medium | 2 | Light–Medium |
| `fast1080p` | 1080p | H.264 | fast | 2 | Medium |
| `hevc1080p` | 1080p | HEVC | medium | 4 | Heaviest |

- **Resolution:** 1920×1080 (1080p)
- **Frame Rate:** 30fps
- **Codec:** H.264 (`libx264`)
- **Bitrate:** ∼8 Mbps (ABR)

To introduce realistic complexity, as different video contents have different compression characteristics, we generated videos with five distinct visual patterns (*gradient*, *noise*, *moving*, *mandelbrot*, *plasma*). These patterns introduce heterogeneity in resource demand, as computationally complex patterns (like '*noise*' or '*mandelbrot*') require significantly more CPU time to transcode than simpler ones (like '*gradient*'), even for the same duration. Furthermore, to create tasks of varying lengths, videos were generated in four different durations: 30, 60, 120, and 180 seconds. This process resulted in a pool of 20 unique source videos (5 patterns × 4 durations) to be used as inputs for the transcoding tasks.

**Task Composition and Heterogeneity**

The complete workload is composed of 100 individual transcoding tasks, designed to be heterogeneous in terms of computational intensity and resource requirements. We defined four distinct task types, as detailed in Table 5.1.

Our workload composition is designed to ensure significant heterogeneity by mixing tasks across different durations, resolutions, codecs, and content patterns. This diverse mix is critical, as Lottarini et al. highlight that realistic benchmarking of video transcoding should consider videos across different resolutions and content complexities, as limited diversity prevents capturing real-world workload characteristics (24). Therefore, the 100 tasks are distributed among the four types with a balanced ratio of 3:3:2:2, resulting in: 30 `light1c` tasks, 30 `medium480p` tasks, 20 `fast1080p` tasks, and 20 `hevc1080p` tasks.

**Workload Duration and Submission Pattern**

The mix of four different source video durations naturally creates a workload with a heterogeneous execution time profile, which is crucial for testing the scheduler's ability to handle both short and long-running jobs.

The task arrival process was designed to simulate a periodic, bursty submission pattern, which is common in batch processing environments. The 100 tasks are submitted in 10 consecutive bursts. Each burst contains 10 tasks and arrives every 300 seconds (5 minutes). Within each burst, the 10 tasks are submitted at one-second intervals. To ensure the reproducibility of the experiment, the specific order of tasks assigned to these submission slots is deterministically controlled by the random seed `20250901`. This submission model creates periodic contention for resources, providing a rigorous test for the scheduling and resource management policies under evaluation.

**Trace Generation**

This workload is deployed and executed on the Continuum physical platform described in Section 5.1.1. During execution, our framework's data collection pipeline captures the lifecycle events (submission, start, and end timestamps) and resource consumption metrics (CPU, memory usage) for each transcoding task. This collected data is then processed into the Core Trace Schema (CTS) artifacts that drive the OpenDC simulator.

### 5.1.3 Configuration for Input Fidelity Validation

The initial experiment, which establishes the ground truth for he input fidelity validation, is conducted using a specific, well-defined scheduling policy. This policy dictates how tasks from the workload are assigned to the virtual machines on the execution platform.

For our baseline, we employ a Centralized, Strict FIFO Push Scheduler. This initial policy represents a simple and deterministic approach to managing batch tasks. Its behavior is defined by the following rules:

- **Centralized Queue:** A single, central queue holds all submitted tasks. Tasks are placed in this queue in the exact order of their arrival.
- **Strict FIFO Selection:** Tasks are selected from the head of the queue in a strict First-In, First-Out (FIFO) order for dispatching. No other factors, such as task length or resource requirements, are considered.

- **Single-Task Concurrency per Node:** Each worker node (virtual machine) can only execute a single task at a time. A node is considered "busy" for the entire duration of a task's execution and only becomes "idle" upon its completion.
- **Centralized Push and Sorted Node Selection:** A central scheduler entity is responsible for dispatching tasks. When the task at the head of the queue is ready to be dispatched, the scheduler first identifies the set of all worker nodes that are currently idle. It then sorts these idle nodes by their names in lexicographical order. The task is "pushed" to the first node in this sorted list (i.e., the one with the lexicographically smallest name).

To ensure the simulation is a faithful counterpart for the input fidelity validation, we configured OpenDC to precisely replicate the baseline behavior:

- **Scheduler and FIFO Logic:** We use the `FilterScheduler` provided by OpenDC, which processes the scheduling queue sequentially to ensure a strict FIFO ordering of tasks.
- **Concurrency Constraint:** The "single task per node" rule is enforced by including an `InstanceCountFilter` in the scheduler's filter chain, with its `limit` parameter set to `1`.
- **Node Selection Logic:** To mirror the "select the lexicographically smallest" rule, we ensure that the hosts in our topology definition file are listed in the lexicographical order of their names. This guarantees that the `FilterScheduler`, which defaults to picking the first available host, replicates the physical policy's logic.

This policy was chosen for the baseline due to its simplicity, determinism, and prevalence as a standard benchmark in scheduling system analysis. The trace data collected under this configuration is used to conduct the input fidelity validation of the OpenDC simulator in Section 5.2. Furthermore, this policy will be included in our Design Space Exploration in Section 5.3, serving as a crucial control group to quantify the improvements offered by more advanced scheduling strategies.

### 5.1.4 Design Space Exploration (DSE) Configuration

To systematically investigate the performance and energy characteristics of different management strategies, we define a design space for exploration within the validated OpenDC

simulator. This exploration is conducted using the same workload as the input fidelity validation, which is detailed in Section 5.1.2.

The design space is constructed by combining multiple levels across two main factors: Cluster Topology and Scheduling Policy Configuration. We defined five base scheduling policies, representing different placement philosophies. Three of these policies were tested across four levels of CPU overcommitment, while the two simpler baseline policies were tested only without overcommitment. This approach allows us to analyze not only the policies themselves but also their behavior under different resource pressure conditions. These exploratory factors align with established engineering practices. Cloud platforms commonly employ CPU resource overcommitment to enhance overall utilization, using overload management to maintain SLAs (25). Concurrently, consolidation (bin-packing) is used to significantly improve utilization and cluster throughput without violating latency objectives (26). As a counterpoint to consolidation, production-grade schedulers also widely provide spread constraints (distributing tasks across hosts) to mitigate the risk of correlated failures and reduce tail latency, which corresponds directly to the 'spread' policies included in our DSE (27).

The full set of factors and levels is detailed in Table 5.2.

The combination of these factors and levels results in a total of 14 unique scheduling configurations (2 baseline policies + 3 advanced policies × 4 overcommitment ratios). When combined with the 2 cluster topologies, we arrive at 28 distinct experimental scenarios.

To ensure the stability of the results and account for any minor sources of non-determinism in the simulation, each of these 28 scenarios is run three times, using three distinct initial seeds (0, 1, and 2). This leads to a total of 84 simulation runs. All policies are built upon OpenDC's `FilterScheduler` with `subsetSize=1` to ensure deterministic task processing and host selection within a single run. The results of this comprehensive exploration, aggregated across the different seeds, are presented in Section 5.3.

### 5.1.5 Evaluation Metrics

To quantitatively evaluate the outcomes of both the input fidelity validation (Section 5.2) and the design space exploration (Section 5.3), we define two distinct sets of metrics. The first set is used in our input fidelity validation to assess the fidelity of the simulator against the physical ground truth. The second set consists of high-level Key Performance Indicators (KPIs) used for the comparison and ranking of different policies in our design space exploration.

**Table 5.2:** Factors and levels for the DSE.

| Factor | Levels |
|---|---|
| Cluster Topology | • 4x4: 4 VMs, each with 4 vCPUs<br>• 2x8: 2 VMs, each with 8 vCPUs |
| Base Scheduling Policy & CPU Overcommitment Ratio | 1. Baseline Policies (Ratio = 1.0 only):<br>• `Baseline-IC1`: Enforces no co-location (max 1 instance per host).<br>• `FirstFit`: A simple sequential-fit policy.<br>2. Advanced Policies (Ratio = 1.0, 1.25, 1.5, and 2.0):<br>• `Pack-IC+`: A consolidation policy that prefers hosts with more running instances.<br>• `Spread-IC-`: A spreading policy that prefers hosts with fewer running instances.<br>• `Spread-vCPU`: A load-balancing policy that prefers hosts with the most available vCPU capacity. |

#### 5.1.5.1 Metrics for Input Fidelity Validation

These metrics are designed to compare the dynamic behavior of the simulation against the physical execution trace, ensuring the simulator is a trustworthy model.

**Task Wait Time Distribution**

This metric evaluates the fidelity of the simulator in reproducing the queuing dynamics of the physical system. We assess this by comparing the entire statistical distribution of task wait times (the time from submission to the start of execution). The detailed calculation method, the Kolmogorov–Smirnov test, and the specific quantile error thresholds used are defined in Sections 4.4.2 and 4.4.3.

**Cumulative Completion Curve**

This metric measures the alignment of the overall task processing progress and throughput over time. Consistency is assessed by the normalized Root Mean Square Error (RMSE%) between the curves from the physical and simulated runs. The specific implementation, including lag-optimization, is detailed in Sections 4.4.2 and 4.4.3.

**Instantaneous CPU Usage**

Instantaneous CPU Usage This metric provides a microscopic validation of the workload trace transformation by comparing the fine-grained CPU usage. Consistent with the OpenDC simulation model, this "usage" is quantified in MHz, representing the absolute processing power consumed by the task rather than a relative utilization percentage. The comparison is performed on a per-task basis using a relative time axis, as detailed in Section 4.4.2. This approach is designed to specifically validate the intrinsic behavioral profile of each task, isolating the model's fidelity from minor scheduling-induced time shifts. The precise error metrics, such as SMAPE and Pearson correlation, are specified in Section 4.4.3.

**Total Power Draw Trend**

This metric assesses the fidelity of the simulator's energy model by comparing the shape and synchronicity of the total power consumption curve against the physical measurements. The comparison focuses on the trend, measured by the Pearson correlation coefficient, rather than absolute values. The data reconstruction and comparison methodology are described in Sections 4.4.2 and 4.4.3.

#### 5.1.5.2 Design Space Exploration and Decision-Making Metrics

Once the simulator's baseline fidelity is established, we use a set of high-level KPIs to evaluate and rank the different policy configurations. These are divided into primary objective metrics, which guide the Top-$k$ selection process, and secondary diagnostic metrics, which provide deeper insights into the behavior of each policy.

**Primary Objective Metrics**

These two KPIs represent the core trade-off between performance and energy efficiency. They will be used to construct the Pareto frontier for identifying the Top-$k$ candidate policies.

- **Total Makespan:** The total time elapsed from the submission of the first task to the completion of the last. This is the primary indicator of overall system throughput.
- **Energy per Task:** The average energy consumed to complete a single task, calculated by dividing the total energy consumption by the number of completed tasks. This is the primary metric for energy efficiency.

**Secondary Diagnostic Metrics**

These metrics will be used to conduct a more nuanced analysis of the candidate policies and to explain the underlying reasons for their performance.

- **Task Turnaround Time (P95 and Average):** The total time a task spends in the system, from its submission to its completion. We measure both the average and the 95th percentile (P95) to evaluate overall efficiency and the predictability of task completion times.
- **Task Wait Time (P95 and Average):** The time a task spends in the queue before its execution begins. The average and P95 values are used to assess the scheduler's responsiveness and fairness.
- **Average CPU Utilization:** The average utilization of all CPU cores across all nodes over the makespan. This metric provides insight into the resource efficiency of a given policy.

The implementation of the automated analysis scripts that calculate all these KPIs from the simulation output is described in Section 4.4.5.

## 5.2 Input Fidelity Validation

This section presents the results of our input fidelity validation, a critical step to establish trust in the simulation model before using it for broader exploration. The primary objective is to quantitatively assess the fidelity of the OpenDC simulator by comparing its output against the ground-truth data collected from the physical execution platform. To provide a comprehensive assessment, our analysis will sequentially examine four key metrics: the task wait time distribution, the cumulative task completion curve, the instantaneous CPU usage, and the total power draw trend.

### 5.2.1 Task Wait Time Distribution

The first metric we evaluate is the task wait time distribution, which assesses the simulator's ability to accurately reproduce the queuing dynamics of the physical system. A close match in this metric indicates that the simulator's scheduling logic and its modeling of resource contention are faithful to reality.

Figure 5.1 presents the Cumulative Distribution Function (CDF) of task wait times. In the top plot, the curves for the physical execution (Continuum) and the simulation (OpenDC) show a close visual correspondence, indicating a high degree of alignment. This high fidelity is expected, as the input fidelity validation uses a simple, deterministic FIFO scheduler (one task per node) which the simulator was configured to precisely replicate.

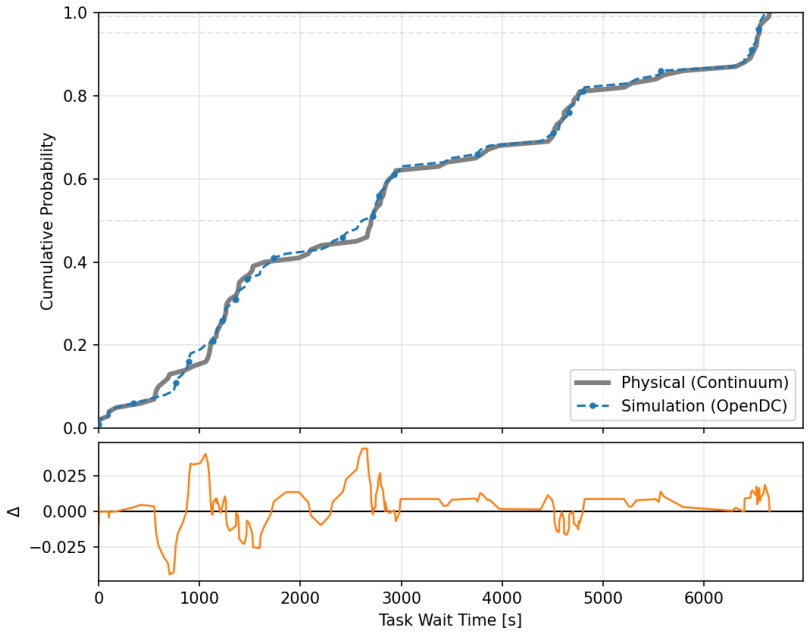


**Figure 5.1:** Cumulative Distribution Function (CDF) of task wait times, comparing the physical and simulated results (top) and their residual difference (bottom).

This alignment thus confirms the simulator's core contention model mirrors the physical system's behavior under these controlled conditions. To analyze the minor deviations, a residual subplot is provided at the bottom, which plots the difference in cumulative probability ($\Delta$ = Simulation − Physical). The subplot reveals that the magnitude of the error remains small, with the maximum deviation remaining below 0.03 (3%).

While Figure 5.1 provides a strong visual confirmation of alignment, a rigorous validation must move beyond subjective visual inspection to objective, falsifiable metrics. Therefore, for a formal quantitative assessment, we employ the dual-criterion approach defined in Section 4.4.3. The comparison, summarized in Table 5.3, uses a two-sample Kolmogorov-Smirnov (KS) test for the overall distributional shape and calculates the relative error for key percentiles.

The KS test yields a p-value of 0.99, which is significantly greater than the significance level of 0.05, satisfying the criterion for distributional shape similarity. Furthermore, the relative errors for the p50, p95, and p99 percentiles are 1.56%, 0.19%, and 0.83% respectively. All of these fall well within their pre-defined acceptance thresholds of 5%, 10%, and 20%.

**Table 5.3:** Quantitative validation of the task wait time distribution. The table compares key statistical properties from the physical execution against the simulation. The values for p50, p95, and p99 are the corresponding percentile wait times in seconds. "Rel. Error" stands for Relative Error, calculated between the simulated and physical percentile values.

| **Validation Criterion** | **Physical (Continuum)** | **Simulation (OpenDC)** | **Comparison Metric** | **Outcome (vs. Threshold)** |
|---|---|---|---|---|
| KS Test (p-value) | – | – | 0.99 | PASS ($\geq 0.05$) |
| p50 Wait Time | 2706.81 s | 2664.59 s | 1.56% (Rel. Error) | PASS ($\leq 5.0\%$) |
| p95 Wait Time | 6545.11 s | 6532.43 s | 0.19% (Rel. Error) | PASS ($\leq 10.0\%$) |
| p99 Wait Time | 6640.28 s | 6585.33 s | 0.83% (Rel. Error) | PASS ($\leq 20.0\%$) |

Given that both the visual representation, the distributional shape test, and the key percentile points meet our consistency criteria, this metric passes the validation check. This result provides strong evidence that our simulation accurately models the task queuing behavior of the baseline scheduling policy.

### 5.2.2 Cumulative Completion Curve

Having validated the fidelity of the system's queuing dynamics, we next evaluate the cumulative completion curve to assess the macroscopic performance alignment between the simulation and the physical system. This metric is crucial for verifying that the simulator accurately models the overall system throughput and the final makespan (total completion time).

Figure 5.2 displays the cumulative number of completed tasks over time. In the top plot, the curves for the physical and simulated runs are almost superimposed, indicating a strong agreement in the rate of task completion throughout the experiment. The residual subplot, which shows the difference in completed tasks ($\Delta$ = Simulation − Physical) at any given moment, confirms this high fidelity. The difference between the two systems rarely exceeds two tasks, and any minor lag is quickly resolved, with the plot hovering consistently around zero.

The quantitative validation, summarized in Table 5.4, focuses on both final makespan accuracy and overall curve similarity. The results demonstrate a high degree of fidelity: the relative error in the final makespan is a mere 0.13%, below the 1.0% acceptance threshold

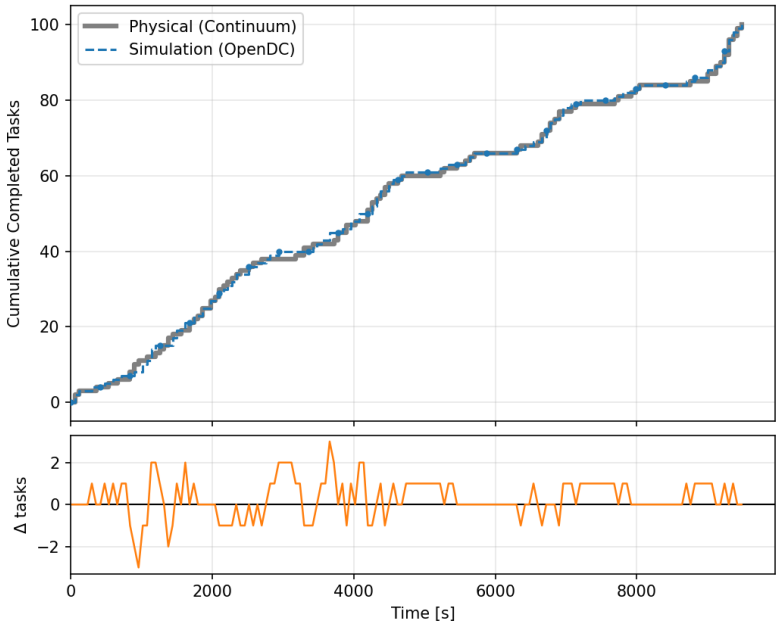


**Figure 5.2:** Cumulative completion curves, comparing the task completion progress for the physical and simulated runs (top) and their residual difference (bottom).

**Table 5.4:** Quantitative validation of the cumulative completion curve. The table compares the total makespan and the Root Mean Square Error (RMSE%) between the physical and simulated runs.

| **Validation Criterion** | **Physical (Continuum)** | **Simulation (OpenDC)** | **Comparison Metric** | **Outcome (vs. Threshold)** |
|---|---|---|---|---|
| Makespan | 9480.41 s | 9492.311 s | 0.13% (Rel. Error) | PASS ($\leq 1.0\%$) |
| Curve Similarity (RMSE%) | – | – | 0.94% | PASS ($\leq 3.0\%$) |

for long-running workloads. Similarly, the overall curve similarity is excellent, with a best-lag Root Mean Square Error (RMSE%) of only 0.94%, also well within the 3.0% threshold.

With both key criteria being met by a significant margin, this metric passes the validation check. This result confirms that our simulation framework accurately models the macroscopic throughput and total runtime of the system under the baseline configuration.

**Table 5.5:** Quantitative validation of instantaneous CPU usage. The table presents key metrics assessing the alignment between the physical measurements and the simulated workload trace.

| Validation Criterion | Comparison Metric | Outcome (vs. Threshold) |
|---|---|---|
| sMAPE | 2.67% | PASS ($\leq 5.0\%$) |
| Pearson Correlation (r) | 0.982 | PASS ($\geq 0.95$) |
| Relative RMSE (% of Median) | 5.69% | PASS ($\leq 10.0\%$) |

### 5.2.3 Instantaneous CPU Usage

With the macroscopic performance alignment confirmed, we now perform a microscopic validation of our trace transformation pipeline. This metric directly assesses the fidelity of the process that converts raw resource consumption data into the fine-grained workload fragments. As per our methodology, this comparison is performed using a per-task, relative time alignment to ensure we are validating the task's intrinsic behavior, rather than artifacts of scheduling jitter.

Figure 5.3 visualizes this comparison using a scatter plot. Each time-aligned data pair is represented as a single point, mapping the physical (Continuum) measurement to the X-axis and its corresponding simulated (OpenDC) value to the Y-axis. As defined in Section 5.1.5.1, both axes quantify this demand in MHz, representing the absolute processing power consumed rather than a utilization percentage. The resulting plot demonstrates a strong linear relationship, with the vast majority of points tightly clustered around the $y = x$ identity line. This indicates a high degree of correlation between the measured physical CPU usage and the usage specified in the simulation trace. We observe a particularly dense cluster around the 4500 MHz mark, showing excellent accuracy for mid-range loads. While there is increased variance in the high-load region (near 9600 MHz), a characteristic artifact of modeling CPU saturation, the overall trend remains highly consistent. The quantitative assessment, based on the criteria in Section 4.4.3, is summarized in Table 5.5. We evaluate three key error and correlation metrics. All three quantitative criteria are met with a significant margin. The sMAPE of 2.67% and the Relative RMSE of 5.69% are both well below their respective thresholds, indicating a low overall error. The Pearson correlation coefficient of 0.982 confirms the extremely strong positive linear relationship observed in the scatter plot.

Given the strong visual correlation and the successful outcome across all three quantitative checks, this metric passes the validation. This is a critical result, as it provides strong

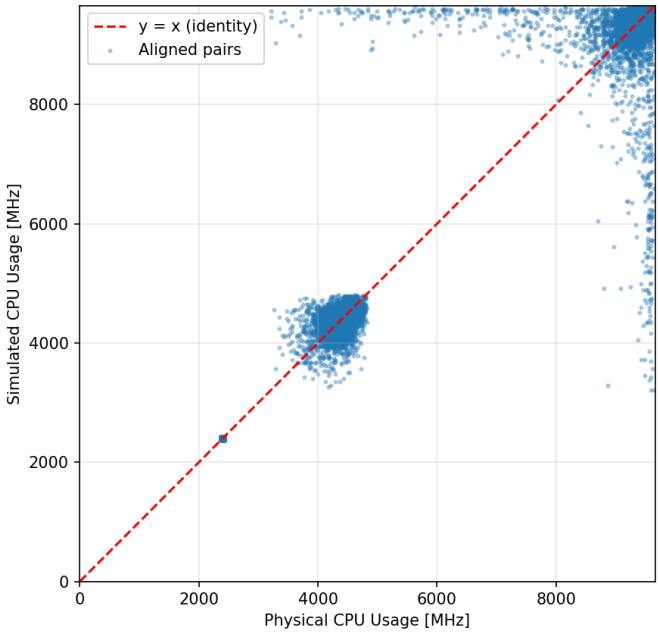


**Figure 5.3:** Scatter plot of instantaneous CPU usage. Each point (dot) represents a single time-aligned (Physical, Simulated) data pair, comparing physical measurements (X-axis) against the simulation trace (Y-axis). The proximity to the $y = x$ identity line indicates high fidelity.

evidence for the fidelity of our trace-driven methodology: the ability to faithfully convert raw, real-world measurements into a representative simulation workload at a fine-grained level.

### 5.2.4 Total Power Draw Trend

Having confirmed the fidelity of the simulation at the microscopic CPU level, the final validation metric assesses the simulator's energy model, whose behavior is primarily driven by this CPU activity. To conduct a comprehensive and robust assessment, we evaluate the dynamic trend consistency using our segmented methodology.

Figure 5.4 presents the Z-score normalized power trend curves from both the physical and simulated platforms. A strong visual correlation is immediately apparent. The simulated power curve closely tracks the peaks and valleys of the physical measurements, successfully capturing the power spikes corresponding to the bursty arrival of tasks defined in our workload. This provides strong qualitative evidence of the model's fidelity. To move beyond visual inspection and perform a rigorous quantitative assessment, we applied the *Segmented Correlation Assessment* methodology. The experiment's duration was partitioned into 32

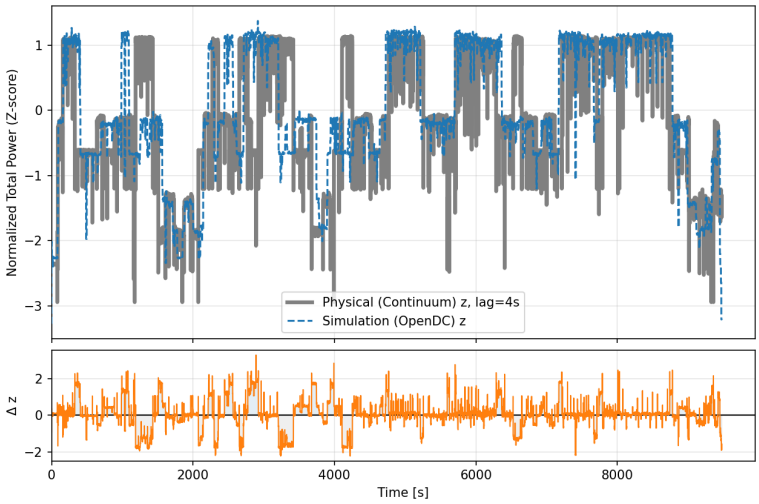


**Figure 5.4:** Z-score normalized total power draw trends, comparing the physical and simulated results (top) and their residual difference (bottom).

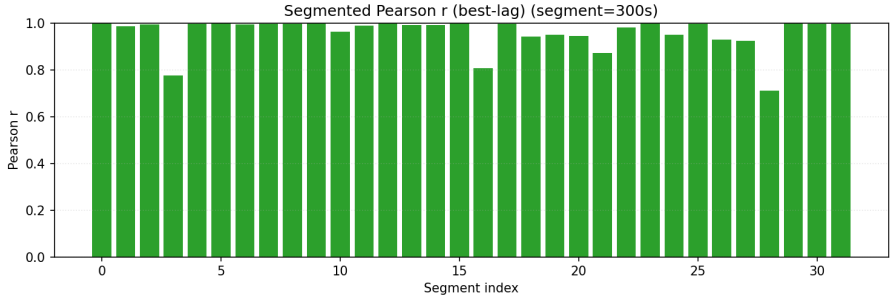


**Figure 5.5:** Segmented Pearson correlation of total power draw trends. The chart displays the best-lag Pearson correlation coefficient (r_k) for each 300-second segment, comparing the Z-score normalized power draw of the physical (Continuum) and simulated (OpenDC) platforms.

segments of 300 seconds each. Figure 5.5 5.5 visualizes the result of this analysis, plotting the best-lag Pearson correlation coefficient ($r_k$) for each segment. The quantitative results, summarized in Table 5.6, are outstanding. Our primary validation metric, the average segmental correlation, reaches an impressive 0.958, which decisively surpasses the predeclared threshold of 0.90. As visually confirmed by Figure 5.5, the fidelity is not just high on average, but remarkably consistent across the experiment's lifecycle. The chart further reveals that even in the segments with the lowest scores, the correlation remains strong, demonstrating the model's robustness under varying workload conditions. Given that the

**Table 5.6:** Quantitative validation of the total power draw trend using the segmented method.

| **Validation Criterion** | **Comparison Metric** | **Outcome (vs. Threshold)** |
|---|---|---|
| Mean Segmental Correlation (Pearson r) | 0.958 | PASS ($\geq 0.9$) |

**Table 5.7:** Summary of Input Fidelity Validation Outcomes.

| **Section** | **Validation Metric** | **Key Result** | **Outcome** |
|---|---|---|---|
| 5.2.1 | Task Wait Time Distribution | KS p-value = 0.99; p99 error = 0.83% | PASS |
| 5.2.2 | Cumulative Completion Curve | Makespan error = 0.13%; RMSE% = 0.94% | PASS |
| 5.2.3 | Instantaneous CPU Usage | Pearson r = 0.982; sMAPE = 2.67% | PASS |
| 5.2.4 | Total Power Draw Trend | Pearson r = 0.958 | PASS |

primary quantitative criterion was met with a significant margin, and this result is strongly supported by visual evidence of both qualitative alignment and sustained consistency, this metric decisively passes the validation check. This result establishes the simulator as a reliable tool for conducting the energy-related analysis in the subsequent design space exploration.

### 5.2.5 Input Fidelity Validation Summary

This section presented a comprehensive, multi-faceted input fidelity validation of our trace-driven simulation framework against its physical counterpart. The fidelity of the simulation was rigorously assessed across four key metrics, each targeting a different aspect of system behavior, from microscopic resource usage to macroscopic performance and energy trends.

The results of this validation are overwhelmingly positive. A summary of the outcomes for each metric is provided in Table 5.7.

As demonstrated, the simulation passed all pre-declared consistency criteria with a significant margin. The framework proved capable of accurately reproducing:

- The queuing dynamics of the system, showing that the scheduling logic is correctly modeled.

- The macroscopic throughput and makespan, confirming the model's ability to predict overall performance.
- The fine-grained CPU behavior, validating the core of our trace transformation pipeline.
- The dynamic power consumption trends, establishing the reliability of the energy model.

In conclusion, the Input Fidelity Validation is successful. We have established a high degree of confidence that our simulation model is a faithful and trustworthy representation of the physical system for the given workload and configuration. This successful input fidelity validation provides the necessary foundation to proceed with the next phase of our evaluation: the simulation-driven Design Space Exploration.

## 5.3 Simulation-Driven Design Space Exploration

Having successfully established the baseline fidelity of our simulation model in the previous section, we now use this trustworthy tool to perform a broad Design Space Exploration (DSE). The objective of this section is to systematically analyze the results of the 22 experimental scenarios defined in Section 5.1.4, in order to understand their performance and energy characteristics and to identify the most promising policy configurations.

This analysis is presented in two parts. First, in Section 5.3.1, we provide a global analysis of the entire design space, visualizing the results to identify key trends and the Pareto frontier. Subsequently, in Section 5.3.2, we perform a detailed comparison of the top-performing policies and select a small set of Top-k candidates for the final output fidelity validation phase.

### 5.3.1 Performance and Energy Analysis

Following the successful input fidelity validation of the simulator, we now use it to perform the design space exploration defined in Section 5.1.4. This section presents and analyzes the results of the 84 simulation runs (28 distinct experimental scenarios, each repeated three times with different random seeds), aggregated by their experimental scenario, to understand the impact of cluster topology, scheduling policy, and CPU overcommitment on performance and energy.

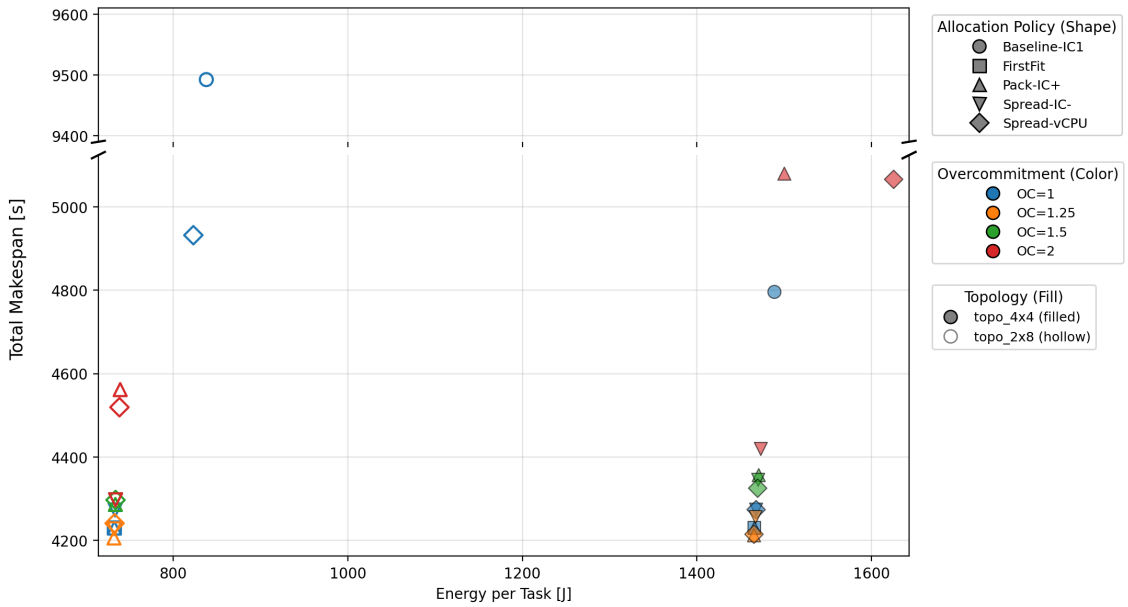


**Figure 5.6:** Performance-energy scatter plot of experimental scenarios.

Figure 5.6 presents the primary results of our exploration in a performance-energy scatter plot. The two inset plots provide magnified views of the two main data clusters, which, as our analysis reveals, correspond directly to the two different topologies.

Our primary observation from the plot is a distinct separation of the results into two groups based on the cluster topology. The Topology 2x8 configurations (hollow markers) form a cluster in the desirable bottom-left region, while the Topology 4x4 configurations (solid markers) form a separate cluster in the top-right. This demonstrates a clear and significant finding: for this workload, the "scale-up" approach of using fewer, larger nodes (2x8) is Pareto-superior to the "scale-out" approach (4x4). A detailed comparison of the data reveals that, for the same scheduling policy, switching from the 4x4 to the 2x8 topology results in a marginal improvement in Total Makespan and, more dramatically, reduces Energy per Task by nearly 50%.

Within each topological cluster, the scheduling policies exhibit complex and nuanced performance characteristics. Focusing on the superior topo_2x8 topology, a key finding emerges: the consolidation policy (Pack-IC+) and the simple baseline (FirstFit) are highly competitive, particularly at lower overcommitment ratios. In fact, Pack-IC+ at a 1.25 overcommitment ratio achieves the lowest makespan not only within this topology but across all 28 experimental scenarios. This performance is slightly better than the spreading policies (Spread-IC- and Spread-vCPU), which, while also efficient, do not consistently

surpass the top performers in terms of throughput. However, the relative effectiveness of the policies can shift under heavy resource pressure. For instance, in the topo_4x4 cluster with a 2.0x overcommitment ratio, the Spread-IC- policy significantly outperforms Pack-IC+. This suggests that while a well-managed consolidation strategy can be surprisingly effective for minimizing total runtime, its advantage is not universal and can be reversed under high levels of resource overcommitment.

The effect of CPU overcommitment reveals a more nuanced trend. For the advanced policies, a moderate overcommitment ratio of 1.25x appears to be the "sweet spot" across both topologies, delivering an optimal balance of Makespan and Energy per Task. Increasing the ratio beyond this point generally leads to a degradation in both performance and energy efficiency. This indicates a clear point of diminishing returns, where excessive resource pressure from co-located tasks begins to negatively impact the system's overall efficiency, an effect particularly noticeable at the 2.0x ratio.

Overall, the policies that constitute the Pareto frontier of our design space are exclusively configurations running on the superior Topology 2x8, primarily at the 1.25x overcommitment level. The best-performing policies, which cluster tightly in the optimal bottom-left corner of the plot, are the consolidation strategy (Pack-IC+) and the simple FirstFit baseline.

### 5.3.2 Selection of Top-k Candidate Policies

The design space exploration in the previous section allows us to identify a set of Top-k candidate policies based on our performance and energy criteria. From this set, we now highlight three notable candidates for a deeper inspection. These configurations represent different optimization goals: the best overall throughput and efficiency, the best tail latency for quality of service, and a simple yet highly competitive baseline. In our results, the configuration that yielded the lowest energy consumption was also the one with the best overall makespan. Therefore, our highlighted policies include this top performer, the configuration with the best tail latency (P95 Task Wait Time), and the surprisingly effective FirstFit strategy, which offers a strong balance of performance and simplicity.

The highlighted policies in Table 5.8 reveal important trade-offs and a key insight into datacenter efficiency. Notably, the top performer, Pack-IC+ at a 1.25x overcommitment ratio, achieves both the lowest makespan and the lowest energy consumption. This finding may seem counterintuitive, as consolidation policies are often expected to increase contention. However, for this workload, a well-managed consolidation strategy proves superior for both speed and efficiency. By strategically packing tasks, it likely improves data

**Table 5.8:** Performance and energy metrics of the three highlighted policy configurations from the simulation.

| Candidate Role | Scheduling Policy | OC Ratio | Makespan (s) | Energy/Task (J) | P95 Wait Time (s) |
|---|---|---|---|---|---|
| 1. Best Makespan & Energy | Pack-IC+ | 1.25 | 4205.65 | 732.20 | 816.38 |
| 2. Best P95 Wait Time | Spread-IC- | 2.0 | 4297.19 | 734.05 | 490.79 |
| 3. Balanced Baseline | FirstFit | 1.0 | 4229.52 | 732.72 | 1227.33 |

locality and reduces scheduling overhead, leading to faster task completion. This allows the servers to return to an idle state more quickly—a principle known as "race-to-idle." The reduction in execution time is so significant that it outweighs any potential energy cost from higher density, resulting in lower overall energy consumption.

An interesting contrast is the Spread-IC- policy at a 2.0x ratio. While it is slightly slower in terms of overall makespan, it provides the best tail latency (P95 Task Wait Time) by a significant margin. This demonstrates a classic trade-off between throughput and quality of service (QoS). The higher overcommitment ratio allows the scheduler to start tasks more quickly, thus reducing their time in the queue. However, this comes at the cost of increased contention during execution, which prolongs the total makespan.

Finally, the FirstFit policy emerges as a highly effective balanced baseline. Without any overcommitment, it delivers a makespan and energy efficiency that are nearly identical to the top-performing Pack-IC+ configuration. This highlights that a straightforward, non-overcommitted strategy can be surprisingly competitive, offering excellent throughput and efficiency, though at the expense of higher tail latency compared to more aggressive, QoS-focused policies. This insight is crucial for operators who may need to prioritize predictable performance for individual tasks over raw system throughput.

## 5.4 Output Fidelity Validation and Model Fidelity Analysis

The previous section utilized the validated simulator to conduct a wide-ranging Design Space Exploration, which successfully identified a set of Top-k candidate policies based on their predicted performance and energy trade-offs. While the simulation provides valuable insights, the 'closed-loop' methodology proposed in this thesis is incomplete without the final, crucial step: output fidelity validation.

**Table 5.9:** Performance and energy metrics from the output fidelity validation on Continuum.

| Candidate Role (from Simulation | Scheduling Policy | OC Ratio | Makespan (s) | Energy/Task (J) | P95 Wait Time (s) |
|---|---|---|---|---|---|
| 3. Balanced Baseline | FirstFit | 1.0 | 4358.72 | 298.35 | 1361.72 |
| 2. Best P95 Wait Time | Spread-IC- | 2.0 | 4412.81 | 298.68 | 694.29 |
| 1. Best Makespan & Energy | Pack-IC+ | 1.25 | 4440.66 | 298.78 | 985.44 |

This section is therefore dedicated to performing this critical validation. We deploy the top-performing candidates from simulation back onto the physical (Continuum) platform. The primary objective is two-fold: first, to evaluate the simulator's fidelity in predicting relative performance rankings under diverse configurations; and second, to verify the ultimate 'decision-support value' of our framework by comparing the predicted optimal policies against real-world outcomes.

### 5.4.1 Output Fidelity Validation Results

Following the simulation-based selection, the three candidate policies, Pack-IC+ (OC=1.25), Spread-IC- (OC=2.0), and FirstFit (OC=1.0) were deployed and executed on the physical Continuum platform. The experimental setup, including the topo_2x8 topology and the workload, was configured to be identical to the simulation environment to ensure a valid comparison.

The real-world performance metrics for these policies were collected and are presented in Table 5.9. For consistency with the simulation metrics in Table 5.8, the total measured energy consumption has been normalized to 'Energy per Task'. The policies in the table are sorted by their actual measured Total Makespan.

A preliminary review of these results indicates that FirstFit (OC=1.0) achieved the shortest total makespan, while Spread-IC- (OC=2.0) maintained the best P95 Wait Time. A detailed comparison and analysis of these findings against the simulation predictions will be conducted in the following section.

**Table 5.10:** Comparison of Simulated Predictions and Output Fidelity Validation Results.

| Metric and Ranking | Scheduling Policy | OC Ratio | Sim. Value | Phys. Value | Sim. Rank | Phys. Rank |
|---|---|---|---|---|---|---|
| **Total Makespan (s)** *(Lower is better)* | Pack-IC+ | 1.25 | 4205.65 | 4440.66 | 1 | 3 |
| | FirstFit | 1.0 | 4229.52 | 4358.72 | 2 | 1 |
| | Spread-IC- | 2.0 | 4297.19 | 4412.81 | 3 | 2 |
| **Energy per Task (J)** *(Lower is better)* | Pack-IC+ | 1.25 | 732.21 | 298.78 | 1 | 3 |
| | FirstFit | 1.0 | 732.72 | 298.35 | 2 | 1 |
| | Spread-IC- | 2.0 | 734.06 | 298.68 | 3 | 2 |
| **P95 Wait Time (s)** *(Lower is better)* | Spread-IC- | 2.0 | 490.79 | 694.29 | 1 | 1 |
| | Pack-IC+ | 1.25 | 816.38 | 985.44 | 2 | 2 |
| | FirstFit | 1.0 | 1227.33 | 1361.72 | 3 | 3 |

### 5.4.2 Ranking Consistency Analysis

To rigorously evaluate the simulator's 'decision-support value', we now directly compare the predicted results from simulation (Table 5.8) with the actual results from the output fidelity validation (Table 5.9). This head-to-head comparison is presented in Table 5.10, which includes the absolute values and, most importantly, the relative performance rankings for each key metric.

The comparison in Table 5.10 reveals two critical and divergent findings.

First, the simulation demonstrated positive fidelity in predicting Quality of Service (QoS) rankings. For the P95 Task Wait Time, the simulator successfully predicted that Spread-IC- (OC=2.0) would provide the best tail latency and correctly identified the exact relative ranking of all three policies.

Second, and more importantly, the simulation failed to predict the correct relative ranking for both Total Makespan and Energy per Task. The policy predicted to be the fastest and most energy-efficient (Pack-IC+) was, in reality, the slowest and least efficient of the three. The true optimal policy for both of these core metrics was the non-overcommitted FirstFit baseline, which the simulator had ranked second.

This critical discrepancy in the core throughput and efficiency metrics demands a dedicated analysis. The fact that the simulator, while accurate in input fidelity validation and QoS prediction, failed to correctly rank the optimal policies for makespan and energy strongly suggests a key limitation in its underlying model. The root cause of this predictive failure will be analyzed in detail in the following section.

### 5.4.3 Analysis of Concurrency Modeling Limitations

The root cause of the ranking failure identified in Section 5.4.2 lies in a fundamental limitation of the OpenDC simulation model: its simplistic representation of task concurrency. The different allocation strategies and overcommitment ratios explored in the DSE create varied and complex concurrency scenarios, for which OpenDC lacks a model that accurately reflects real-world process execution.

Internally, OpenDC operates on a fixed-work model, where total work is constant and time is linearly scaled by supply (i.e., Total Work = Demand × Duration; Work Completed = Supply × Time). This model does not account for the additional work or time overheads generated by parallel task execution. Consequently, in the simulation, different parallel execution strategies do not alter the total amount of work to be done.

In a real physical system, however, this assumption does not hold. Parallel execution introduces significant overheads that increase the total "work" (total CPU-seconds) and time required to complete the same set of tasks. The simulation's failure to model these overheads led to its optimistic and incorrect predictions. These real-world overheads are a composite of several factors:

- **Work Inflation:** Extra work-time required for a multi-threaded execution to complete the same amount of work as its sequential equivalent, often amplified as concurrency increases (28) (29). Typical sources include contention in the memory hierarchy (e.g., non-local access in NUMA architectures and bandwidth limitations), cache invalidation and coherence maintenance, bus/interconnect contention, false sharing, and communication/synchronization costs (e.g., atomics, barriers) (28) (29).

- **Idle Time:** Time spent by threads waiting rather than performing useful work, caused by task unavailability, load imbalance, or unresolved dependencies (29).

- **Parallelization Overheads:** Software-level costs associated with managing parallelism, including:

  - *Algorithmic Overheads:* Additional work inherent to the parallel algorithm itself compared to a sequential baseline (e.g., data partitioning, merging, rearrangement) (28).

  - *Scheduling Overheads:* Costs from the OS or runtime, such as task/thread creation, load balancing, and synchronization; finer-grained tasks tend to increase this overhead (28) (29).

By not accounting for these critical overhead factors—particularly *work inflation* from hardware contention—the OpenDC model systematically underestimated the true cost and runtime of the high-concurrency, overcommitted policies. In the simulation, Pack-IC+ (OC= 1.25) was predicted to be fastest because the model only saw the benefits of high resource utilization, not the costs of that utilization. In reality, the FirstFit (OC= 1.0) policy, with its simpler, non-overcommitted concurrency profile, suffered less from these real-world overheads and thus achieved a superior makespan and energy efficiency.

This finding ultimately demonstrates the necessity of the closed-loop methodology. The output fidelity validation step was essential for identifying the fidelity boundaries of the simulation model and correcting a decision that—based on simulation alone—would have been suboptimal.

## 5.5 Evaluation Summary and Implications

This chapter conducted a comprehensive, multi-stage experimental evaluation to answer our third research question regarding the proposed framework's accuracy and decision-support value. The evaluation followed the full closed-loop methodology, proceeding from Input Fidelity Validation (Section 5.2) , to simulation-driven design space exploration (Section 5.3), and culminating in the critical output fidelity validation of the top-ranked policies (Section 5.4).

The evaluation yielded a series of crucial findings. First, the Input Fidelity Validation was successful , demonstrating that the OpenDC simulator could achieve high fidelity against its physical counterpart under a known configuration. Second, the DSE phase successfully used this simulator to explore a wide design space, identifying Pack-IC+ (OC=1.25) as the predicted optimal policy for both makespan and energy efficiency.

The most significant finding, however, emerged from the final output fidelity validation stage. This step revealed a critical discrepancy: the simulator's predictive fidelity failed to hold for the core throughput and energy metrics. The relative rankings for makespan and energy were inverted, with the simulation's top-ranked policy performing as the worst of the candidates in reality .

This discrepancy, rather than invalidating the framework, highlights its necessity and value. Although the output fidelity validation in this use case did not confirm the simulation's optimal rankings for makespan and energy, the closed-loop methodology itself proved its worth. Its value lies in its ability to rigorously test and discover the boundaries of a simulator's fidelity before a flawed decision is deployed. The framework provided the

means to not only identify this predictive failure but also to diagnose its root cause—the simulator's lack of a sophisticated concurrency model (Section 5.4.3). This finding is crucial, as it identifies a non-trivial but solvable modeling challenge that can be addressed in future work. Therefore, the methodology demonstrates its value as a robust decision-support tool, one that prevents the deployment of suboptimal, simulation-derived policies by systematically verifying them against real-world execution.

# 6

# Related Work

This thesis proposes a trace-driven, closed-loop methodology connecting reproducible physical experiments with simulation for cloud datacenter evaluation. To contextualize our work, this chapter reviews relevant literature in two key areas. Section 6.1 discusses trace-driven simulation and workload representation, including common data sources and formats. Section 6.2 covers simulation validation, calibration, and closed-loop methods, relating to Digital Twin concepts. This review highlights existing challenges in trace translation fidelity and systematic model validation, underscoring the need for our proposed Core Trace Schema (CTS) and integrated validation methodology.

## 6.1 Trace-driven Simulation and Workload Representation

Trace-driven simulation employs execution data (traces) from real systems to enhance simulation fidelity compared to synthetic models. This approach benefits from publicly available resources like the large-scale Google cluster traces, which offer detailed job/task data crucial for understanding and modeling datacenter behavior (30). Standardization efforts, such as the Standard Workload Format (SWF) used within the Parallel Workloads Archive (PWA), aim to improve comparability, primarily focusing on HPC workloads by providing a common structure for job submission and completion events (31). Domain-specific simulators also leverage trace data, adapting the methodology for contexts like serverless edge platforms (e.g., FaaS-Sim) or HPC job dispatching simulation (e.g., AccaSim), often using bespoke trace formats tailored to their specific models (32) (33).

Despite these advances, accurately bridging trace data from controlled, reproducible execution platforms (like Continuum) to simulators (like OpenDC) remains a challenge. Public traces or standard formats often lack the necessary granularity or environmental context

matching specific experimental setups, while domain-specific tools sacrifice portability. To address this gap in semantic fidelity and standardization for controlled experimental workflows, this thesis proposes the Core Trace Schema (CTS) and an associated processing pipeline as a consistent interface underpinning our closed-loop evaluation methodology.

## 6.2 Simulation Validation, Calibration, and Closed-Loop Methods

Ensuring simulation trustworthiness requires rigorous validation against physical systems. Pucher et al. demonstrated this by validating a "top-down," perturbation-based simulation of cloud schedulers against a Eucalyptus cloud. Their focus was achieving statistical agreement between simulation and repeated physical runs to build confidence for production deployment, though less emphasis was placed on systematic, iterative calibration based on observed discrepancies (34).

The concept of Digital Twins (DTs) provides a framework for tighter integration. Athavale et al. envision Data Center Digital Twins (DCDTs) as virtual models continuously updated via bidirectional feedback with the physical system, enabling prediction and optimization. Achieving DT fidelity often necessitates explicit calibration loops (35). Wang et al. developed Kalibre, which uses a knowledge-based neural surrogate in an iterative loop to calibrate computationally expensive CFD thermal models against sensor data. This creates a closed loop focused on refining physical model parameters (like airflow rates) using surrogate optimization (36).

While valuable, these approaches leave a gap regarding a systematic methodology to quantitatively validate and iteratively calibrate discrete-event simulators (like OpenDC) against reproducible physical experiments (from platforms like Continuum), especially across multiple fidelity dimensions and assessing final decision-support value. Our work addresses this by proposing such a methodology, featuring defined metrics with consistency criteria, an input-based calibration process, and physical revalidation to bridge reproducible execution with simulation for trustworthy decision support.

# 7

# Conclusion

This thesis addresses key challenges in reliably evaluating and optimizing resource-management policies for cloud datacenters. We argue that while trace-driven simulation is an efficient tool for exploring the design space, its recommendations may be unreliable without an empirical check against both the physical execution platform and the simulation platform. To resolve this, we design, implement, and evaluate a trace-driven, closed-loop evaluation methodology that systematically connects the physical execution platform (Continuum) with the datacenter simulator (OpenDC). Our experimental evaluation shows that this closed-loop approach is beneficial for providing evidence-backed decisions in cloud-datacenter scenarios.

This chapter concludes the study by formally answering the research questions posed in Section 1.3 and by listing the limitations and future work of this thesis.

## 7.1 Answering the Research Questions

**RQ1: How can we design a system centered on the Core Trace Schema (CTS) to systematically bridge the gap between raw execution data from the execution platform and the structured inputs required by the simulator?** We answer this by designing a system centered on a formal Core Trace Schema (CTS)—a standardized, platform-agnostic specification for representing real-world execution traces (Chapter 3). The design of CTS is grounded in an analysis of the semantic gap between the continuous, cumulative metrics available on physical systems and the discrete-event inputs required by simulators. To implement the schema, we design and build a multi-stage data-processing pipeline (Chapter 4). This pipeline reliably transforms raw, multi-source data into the standardized CTS format and subsequently adapts it into simulator-specific workloads,

establishing a robust and reproducible bridge between the execution and simulation platforms.

**RQ2: How can we design a input fidelity validation-and-feedback loop between the execution platform and the simulator to enable validation and rapid design-space exploration?** We answer this by proposing a systematic input fidelity validation-and-feedback loop (Chapter 3). At its core is the construction of a verifiable baseline. We first define a set of Aligned Metric Definitions covering queue dynamics (e.g., task wait time), macroscopic throughput (e.g., cumulative completion curve and total makespan), micro-level resource fidelity (e.g., instantaneous CPU usage), and energy trends (e.g., total power) (Section 3.5.1). We then establish quantitative consistency criteria for each metric (e.g., KS test, RMSE, segmented Pearson correlation) to objectively assess simulation fidelity (Section 4.4.3). Finally, the loop includes a Parameter Calibration Method that systematically adjusts model inputs when discrepancies are detected, thereby iteratively improving the simulator's fidelity to physical reality.

**RQ3: In cloud-datacenter scenarios, what is the impact of the proposed physical /digital-twin methodology on the accuracy of performance and energy evaluation and on its decision-support value?** We answer this comprehensively in Chapter 5.

*On accuracy*, our methodology shows that simulator accuracy is conditional. The evaluation indicates that while the simulator exhibits high fidelity under the baseline configuration (Section 5.2), its accuracy fails in our use case when predicting the relative performance ranking of high-concurrency and oversubscription/overcommit strategies (Section 5.4.2). Thus, the framework's impact lies in providing a systematic way to uncover the simulator's fidelity boundaries.

*On decision-support value*, the methodology demonstrates critical value. The evaluation (Section 5.4) finds that a purely open-loop simulation would lead to an incorrect conclusion (i.e., incorrectly recommending the Pack-IC+ strategy). By enforcing output fidelity validation, our closed-loop approach detects the erroneous recommendation and identifies the true optimal strategy (FirstFit). Moreover, it helps diagnose the root cause of the simulation failure (a missing concurrency model, Section 5.4.3). Therefore, the framework serves as a safeguard against deploying suboptimal decisions, ensuring the robustness of the final decision.

## 7.2 Limitations and Future Work

This section includes limitations and directions for future work of this thesis.

### 7.2.1 Limitations

Despite successfully designing and validating a closed-loop evaluation framework, this study has certain limitations:

- **Model Fidelity.** As analyzed in Section 5.4.3, the core limitation identified—and relied upon for our analysis—lies in OpenDC itself. Its core work model fails to capture real-world performance overheads induced by task concurrency, namely concurrency overheads and work inflation. This leads to inaccuracies in both the ranking and the absolute values when predicting makespan and energy for high-concurrency strategies. This indicates that the decision-support value of our framework is ultimately bounded by the fidelity of the simulator model it integrates.
- **Scope of Evaluation.** Our evaluation focuses primarily on a specific type of workload (CPU-intensive batch jobs represented by FFmpeg video transcoding) and a specific execution environment (the VM-based Continuum platform). Therefore, the effectiveness and applicability of the closed-loop framework to other workload types (e.g., I/O-bound applications, latency-sensitive microservices) or different execution platforms (e.g., bare metal, Kubernetes containers) remain to be further validated.

### 7.2.2 Future Work

The above limitations point to clear and productive directions for future research:

**Modeling Concurrency Overheads.** The most important and urgent direction is to address the model deficiency diagnosed in Section 5.4.3. As discussed in Chapter 5, this is a non-trivial yet solvable modeling challenge. Future work can introduce a more sophisticated concurrency-overhead model into OpenDC (or similar simulators).

One possible path is to implement an empirical *Work Inflation Factor*. For example, the simulator's work model could be modified so that when the state of task concurrency changes, the total work is adjusted to the original work plus the additional overhead induced by concurrent tasks. The output fidelity validation data collected in Section 5.4 can serve as a valuable initial dataset to fit and calibrate this adjustment via regression analysis.

A deeper—though more complex—path is to develop an analytical model that attempts to simulate the sources of work inflation rather than only its symptoms. For instance, one could introduce contention models for shared resources (e.g., last-level cache (L3), memory bus) or quantitative models for operating-system scheduler overheads (e.g., context switches, NUMA migrations). While this path lies beyond the scope of this thesis, it represents a route to making the simulator genuinely trustworthy under high-concurrency scenarios.

**Broaden the Framework's Application.** Future work should apply the closed-loop evaluation framework to a broader set of scenarios to test its generality. This includes:

- **Diverse workloads:** Reproduce the methodology on microservices, big-data analytics (e.g., Spark), and I/O-bound applications to observe how simulation fidelity changes under different resource bottlenecks.
- **Heterogeneous execution environments:** Deploy the framework on bare-metal servers, public-cloud instances, or Kubernetes-orchestrated containers to validate its effectiveness across different levels of virtualization and isolation.

# References


[1] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019. 1

[2] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. **Large-scale cluster management at Google with Borg**. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015. 1

[3] Jeffrey Dean and Luiz André Barroso. **The tail at scale**. *Communications of the ACM*, **56**(2):74–80, 2013. 1

[4] International Energy Agency (IEA). **Energy and AI**. Report, International Energy Agency, Paris, 2025. Licence: CC BY 4.0. 1

[5] Peter Ivie and Douglas Thain. **Reproducibility in scientific computing**. *ACM Computing Surveys (CSUR)*, **51**(3):1–36, 2018. 2

[6] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent Van Beek, and Alexandru Iosup. **OpenDC 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters**. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 455–464. IEEE, 2021. 2, 15, 16, 17

[7] Yi Liang, Nianyi Ruan, Lan Yi, and Xing Su. **An approach to workload generation for modern data centers: A view from Alibaba trace**. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, **4**(1):100164, 2024. 2

[8] Arshdeep Bahga, Vijay Krishna Madisetti, et al. **Synthetic workload generation for cloud computing applications**. *Journal of Software Engineering and Applications*, **4**(07):396, 2011. 2

[9] Guozhi Liu, Weiwei Lin, Haotong Zhang, Jianpeng Lin, Shaoliang Peng, and Keqin Li. **Public Datasets for Cloud Computing: A Comprehensive Survey**. *ACM Computing Surveys*, **57**(8):1–38, 2025. 2

[10] Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah. **{CausalSim}: A causal framework for unbiased {Trace-Driven} simulation**. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1115–1147, 2023. 2

[11] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. **Heterogeneity and dynamicity of clouds at scale: Google trace analysis**. In *Proceedings of the third ACM symposium on cloud computing*, pages 1–13, 2012. 3

[12] Yue Cheng, Zheng Chai, and Ali Anwar. **Characterizing co-located datacenter workloads: An alibaba case study**. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–3, 2018. 3

[13] Muhammad Zakarya, Lee Gillam, Ayaz Ali Khan, and Izaz Ur Rahman. **Perficientcloudsim: a tool to simulate large-scale computation in heterogeneous clouds**. *The Journal of Supercomputing*, 2020. 3

[14] Manoel C Silva Filho, Raysa L Oliveira, Claudio C Monteiro, Pedro RM Inácio, and Mário M Freire. **CloudSim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness**. In *2017 IFIP/IEEE symposium on integrated network and service management (IM)*, pages 400–406. IEEE, 2017. 3

[15] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in the Compute Continuum**. In *Proceedings of the First FastContinuum Workshop, in conjuncrtion with ICPE, Coimbra, Portugal, April, 2023*, 2023. 14, 17

[16] Moysis Symeonides, Zacharias Georgiou, Demetris Trihinas, George Pallis, and Marios D Dikaiakos. **Fogify: A fog computing emulation framework**. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 42–54. IEEE, 2020. 14

[17] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. **Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures**. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6. IEEE, 2017. 14

[18] Jonathan Hasenburg, Martin Grambow, and David Bermbach. **MockFog 2.0: Automated execution of fog application experiments in the cloud**. *IEEE Transactions on Cloud Computing*, **11**(1):58–70, 2021. 14

[19] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. **CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms**. *Software: Practice and experience*, **41**(1):23–50, 2011. 15

[20] Henri Casanova, Arnaud Legrand, and Martin Quinson. **Simgrid: A generic framework for large-scale distributed experiments**. In *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, pages 126–131. IEEE, 2008. 15

[21] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. **Edgecloudsim: An environment for performance evaluation of edge computing systems**. *Transactions on Emerging Telecommunications Technologies*, **29**(11):e3493, 2018. 15

[22] Redowan Mahmud, Samodha Pallewatta, Mohammad Goudarzi, and Rajkumar Buyya. **Ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments**. *Journal of Systems and Software*, **190**:111351, 2022. 15

[23] atlarge-research. **OpenDC: Collaborative Datacenter Simulation and Exploration for Everybody**. `https://github.com/atlarge-research/opendc`, 2025. GitHub repository, accessed 2025-09-10. 17

[24] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler.

**vbench: Benchmarking video transcoding in the cloud**. *ACM SIGPLAN Notices*, **53**(2):797–809, 2018. 60

[25] Salman A Baset, Long Wang, and Chunqiang Tang. **Towards an understanding of oversubscription in cloud**. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012. 63

[26] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. **Heracles: Improving resource efficiency at scale**. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015. 63

[27] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. **Borg, omega, and kubernetes**. *Communications of the ACM*, **59**(5):50–57, 2016. 63

[28] Umut A Acar, Arthur Charguéraud, and Mike Rainey. **Parallel work inflation, memory effects, and their empirical analysis**. *arXiv preprint arXiv:1709.03767*, 2017. 80

[29] Stephen L Olivier, Bronis R De Supinski, Martin Schulz, and Jan F Prins. **Characterizing and mitigating work time inflation in task parallel programs**. *Scientific Programming*, **21**(3-4):123–136, 2013. 80

[30] Charles Reiss, John Wilkes, and Joseph L Hellerstein. **Google cluster-usage traces: format+ schema**. *Google Inc., White Paper*, **1**:1–14, 2011. 83

[31] Dror G Feitelson, Dan Tsafrir, and David Krakov. **Experience with using the parallel workloads archive**. *Journal of Parallel and Distributed Computing*, **74**(10):2967–2982, 2014. 83

[32] Philipp Raith, Thomas Rausch, Alireza Furutanpey, and Schahram Dustdar. **faas-sim: A trace-driven simulation framework for serverless edge computing platforms**. *Software: Practice and Experience*, **53**(12):2327–2361, 2023. 83

[33] Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. **AccaSim: a customizable workload management simulator for job**

**dispatching research in HPC systems**. *Cluster Computing*, **23**(1):107–122, 2020. 83

[34] Alexander Pucher, Emre Gul, Rich Wolski, and Chandra Krintz. **Using trustworthy simulation to engineer cloud schedulers**. In *2015 IEEE International Conference on Cloud Engineering*, pages 256–265. IEEE, 2015. 84

[35] Jyotika Athavale, Cullen Bash, Wesley Brewer, Matthias Maiterth, Dejan Milojicic, Harry Petty, and Soumyendu Sarkar. **Digital twins for data centers**. *Computer*, **57**(10):151–158, 2024. 84

[36] Ruihang Wang, Deneng Xia, Zhiwei Cao, Yonggang Wen, Rui Tan, and Xin Zhou. **Toward data center digital twins via knowledge-based model calibration and reduction**. *ACM Transactions on Modeling and Computer Simulation*, **33**(4):1–24, 2023. 84

# Appendix A

# Reproducibility

## A.1 Abstract

This appendix describes the experimental artifacts developed in this thesis to support reproducibility. The entire closed-loop evaluation framework, including data collection scripts, the CTS processing pipeline, and simulation configurations, is publicly available. The framework is designed to integrate the Continuum execution platform and the OpenDC simulator, and to generate comparable performance and energy metrics . It enables researchers to replicate the validation and exploration results presented in Chapter 5 , and to adapt the methodology for new workloads or simulation models .

## A.2 Artifact check-list (meta-information)

- **Program:** Python, Bash
- **Data set:** Generated by a script
- **Run-time environment:** Ubuntu 22.04.1 LTS, Continuum Framework, Scaphandre, OpenDC
- **Hardware:** x86-64 Linux machine with virtualization support
- **Metrics:** Total Makespan, Energy per Task
- **Output:** JSONL and Parquet workload
- **Experiments:** FFmpeg Batch Processing scenarios
- **How much disk space required (approximately)?:** 10 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 6 to 9 hours
- **Publicly available?:** Yes

## A.3 Description

### A.3.1 How to access

The entire artifact is hosted on GitHub at: `https://github.com/Wsrxgh/Tracekit`. This repository includes the data collection scripts, CTS processing pipeline, and simulation configurations . The repository's README file also provides links to other required tools, such as Continuum and OpenDC.

### A.3.2 Hardware dependencies

The experiments were conducted on a host server running Ubuntu 22.04.1 LTS , equipped with an Intel(R) Xeon(R) Silver 4210R CPU (40 Cores) and 256 GB of memory. The framework requires a Linux host that supports QEMU/KVM virtualization. For replicating the energy measurements, the host CPU must support Intel RAPL, as required by the Scaphandre monitoring tool.

### A.3.3 Software dependencies

The framework's software dependencies vary depending on the experimental stage, such as the physical trace collection on Continuum , the data processing pipeline , or the simulation in OpenDC. Key dependencies include Python (3.8 or newer) , Continuum (version 1.0) , and OpenDC (version 2.4f). A complete list of all software dependencies and their exact versions is provided in the README.md file within the code repository.

## A.4 Installation

As described in the README.md file within the code repository.

## A.5 Experiment workflow

As described in the README.md file within the code repository and Chapter 5.