MOONLIGHT ON THE SERENGETI: CASTING (DIGITAL) SHADOWS WITH SIMBA, PUMBA, LAURA AND FRIENDS

J. Jones*, A. Brynes, M. Johnson, M. King, N. Ziyan
ASTeC STFC Daresbury Laboratory, Daresbury, UK and The Cockcroft Institute, Daresbury, UK

Abstract

Digital shadows offer a methodology for bridging physical particle accelerators with their virtual counterparts, enabling predictive modelling, automated control, and improvements in facility design and implementation. A digital shadow for the CLARA accelerator is being developed, integrating two complementary tools: LAURA, an ontology-driven description language for particle accelerator lattices; and SIMBA, a simulation framework that utilises LAURA-defined models to perform start-to-end particle tracking using multiple tracking codes. Together, LAURA and SIMBA form the simulation backbone of a digital shadow, providing a consistent and extensible representation of the accelerator. This architecture is further augmented by pyCATAP, a controls middle layer that interfaces with accelerator control systems, PUMBA, a procedural automation framework for running complex operational tasks and standardising I/O data structures, and SARABI, a framework for generating soft EPICS IOCs from a LAURA lattice.

INTRODUCTION

The increasing complexity of modern accelerators demands more sophisticated tools for facility design, commissioning, operation and user exploitation. Operators must simultaneously manage hundreds of hardware components, diagnose machine faults, and optimise beam parameters. Computational models have historically been maintained as offline artefacts, somewhat decoupled from the live machine. The digital shadow offers a more tightly integrated approach: a continuously updated virtual representation of the physical accelerator that reflects its current state and can be used to support real-time decision-making whilst providing both real and hidden parameters and diagnostics.

A digital shadow differs from a digital twin in the information flow: a shadow allows only a information flow from the machine to the model; a twin implies a bidirectional coupling with the virtual model not only reflecting the physical system but also issuing commands that directly actuate it. In both cases the ultimate goal is for the model to faithfully track the machine state. For CLARA [1], the development of a digital shadow provides a flexible foundation that can evolve towards fuller twinning as confidence in the underlying tools and models develops.

Realising a digital shadow requires solving several interdependent problems: a lattice must be described in a machine-readable, physics-aware format; simulations must be executable against that description; the live hardware state must be accessible programmatically; and operational procedures must be expressible in a way that can exploit all of the above. We present a suite of complementary, interoperable software packages that addresses each of these requirements.

- **LAURA** (Lattice Architecture for a Unified Representation of Accelerators) provides an ontology-driven description language for particle accelerator lattices, encoding elements, sections, and layouts in human-readable YAML that maps directly to typed Python objects via a Pydantic-based data model [2].
- **SIMBA** (Simulations for Integrated Modelling of Beams in Accelerators) is a simulation framework that uses LAURA-defined lattice models to perform start-to-end particle tracking across multiple simulation codes, enabling rapid propagation of beam distributions through the full machine [2].
- **pyCATAP** (Python Controls Abstraction Towards Accelerator Physics) provides hardware interfacing, and acts as a controls middle layer that abstracts EPICS channel access into high-level Python objects, allowing live machine states to be read and written using the same element vocabulary defined in LAURA [3].
- **PUMBA** (Procedures and Utilities for Measurements on Beam-lines and Accelerators) handles procedural automation, and is a framework for encoding complex operational tasks as composable, reusable procedures that can invoke simulations, acquire measurements, and respond to machine conditions.
- **SARABI** (Soft Architecture for Rendering Automated Back-end IOCs) provides a mechanism for generating soft EPICS Input/Output Controllers (IOCs) directly from a LAURA lattice, enabling virtual signals to be published onto the controls network, alongside real hardware signals, closing the loop between the digital shadow and the broader control-room ecosystem.

Together, these tools form a layered architecture in which a single, ontological lattice description drives both simulation and hardware access, while a common data model ensures that experimental measurements, simulation results, and machine set-points are represented consistently. To this end, we also introduce the CLARA integrated model (see Fig.1), a digital shadow, that acts as a stepping stone towards digital twinning. The integrated model codebase has been built to work on a wide variety of facilities, but the focus for this paper is on CLARA.

In this paper we describe the design of each component, the integration between them, and the current capabilities of the individual components and the digital shadow as a whole [4].

* james.jones@stfc.ac.uk

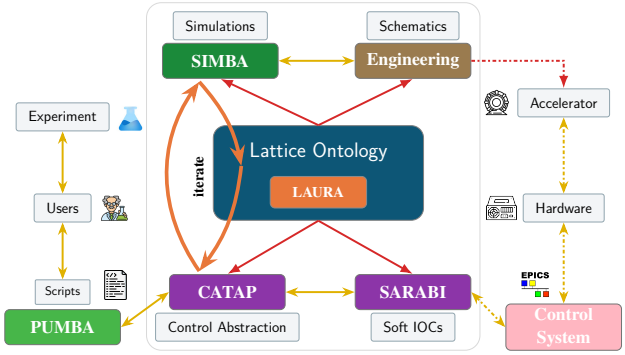

Figure 1: Integrated Model: Machine design and engineering can be iterated using only the *LAURA* lattice description, through the use of *SIMBA* and *PUMBA* to validate procedures and simulated experiments. The virtual control system is automatically created using *SARABI* and integrated with *pyCATAP*. The optimised machine can be built using the mechanical and engineering designs derived directly from the *LAURA* lattice, with the control system architecture extracted from *SARABI*.

COMPONENTS

LAURA: An Ontological Accelerator Language

LAURA provides the descriptive layer of the digital shadow: it defines the machine in a consistent, machine-readable form. LAURA provides a single authoritative description that can be used by multiple downstream tools without rewriting the machine model for each one.

LAURA grew directly out of experience with PAdantic (Particle Accelerator Pydantic model) – a first attempt to demonstrate that accelerator configuration could be expressed in a validated data model. LAURA extends this lineage by formalising the machine description as an ontology: a shared vocabulary of component types, attributes, and relationships that is stable across applications. This shift is important for a digital shadow, as it allows different tools to interpret the same machine definition consistently rather than relying on tool-specific conventions.

The ontology distinguishes physical components (e.g. magnets, diagnostics, etc), their properties (length, strength, position, etc), and their roles in beam transport. It also encodes structural relationships such as containment (elements belonging to sections), ordering (beamline sequence), and machine context (injector, linac, etc). By making these relationships explicit, LAURA enables unambiguous translation between lattice files, simulation codes, and controls software.

LAURA represents the machine through a hierarchy of elements, sections, layouts, and machines:

- *Elements* are the building blocks: individual physical or logical devices that act on the beam or measure it.
- *Sections* are grouped, contiguous parts of the machine that organise elements into meaningful operational regions. Sections can be reused or reconfigured while preserving a consistent element-level definition.
- *Layouts* are complete ordered assemblies of sections that define a full beamline configuration for a given machine state or operating scenario.
- *Machines* are collections of Layouts that describe the full accelerator complex.

LAURA machine definitions are defined as separate repositories and packages, and enable easy integration between LAURA and SIMBA for a choice of machines.

SIMBA: A Start-to-end Simulation Framework

SIMBA is a simulation framework that accepts LAURA-defined lattice models and performs start-to-end (S2E) particle tracking using multiple physics codes working in sequence. SIMBA orchestrates these codes, translating beam distributions between them and maintaining state consistency throughout the simulation. Currently SIMBA supports the following codes with varying levels of implementation: Elegant [5], ASTRA [6], GPT [7], Cheetah [8, 9], Ocelot [10], OPAL [11], Wake_T [12], and Genesis [13].

SIMBA is an evolution of the existing *Accelerator Simulation Framework*, a Python S2E package in use for many years at Daresbury. SIMBA, and LAURA, differ from the original codebase by fundamentally separating the ontological lattice description, the per-code element translation, and the particle tracking and data management aspects of a multi-code S2E simulation from each other.

SIMBA reads in lattice files from LAURA and creates separate lattice lines dynamically based on a definition file. Each section can be tracked in a different tracking code, with SIMBA doing the translating of I/O files and fieldmaps between codes. Lattice elements and section objects are accessible as attributes of the SIMBA instance, controllable via their LAURA names and attributes. Simulation settings, such as the computation of collective effects, initial Twiss parameters, etc., can be configured. Other options, such as for FEL or plasma-based simulations, are also available. Element groups allow coordinated operations, such as the setting of consistent dipole angles and element positions in variable bunch compressors.

The entire lattice, or sections within, can be tracked sequentially (see Fig. 2). An initial beam distribution is either loaded or generated, and both this and the lattice section are converted to the relevant format. After execution, output data are saved in a code-independent HDF5 file format, consistent with the openPMD standard. For computationally-intensive simulations remote execution is possible, with output files transparently returned to the main SIMBA instance. Summary information, such as code-calculated Twiss parameters and output beam distributions, are also produced. Input lattice files, settings, and input beam distributions provide a complete system for later re-running.

The implemented tracking codes currently supported provide a wide range of possible simulation configurations, some of which are incompatible with other supported codes. A reduced set of generic options are currently provided (such as toggling of collective effects; tracking step sizes; beam

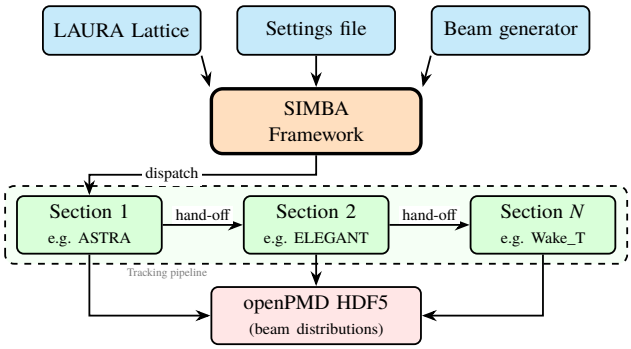

Figure 2: Data-flow diagram of an example SIMBA simulation. Three inputs — a LAURA lattice definition, a settings file, and an initial beam distribution — are read by the SIMBA Framework, dispatched to each lattice section in turn and translating the beam distribution between codes at each hand-off point. Each section produces output stored in the code-independent openPMD HDF5 format; a summary of Twiss parameters and beam statistics is computed across all sections at the end of the run.

matching parameters) but future developments will expand this selection.

pyCATAP: A Package for Controls Abstraction

The pyCATAP package serves as a high-level Pythonic interface designed to decouple beam physics applications from the underlying control system architecture (e.g., EPICS, TANGO). By employing a modular abstraction layer through extensive use of the Pydantic Python library, CATAP translates LAURA definitions for elements into standard Python objects including type-checking and parameter validation. This allows interactions with magnets, diagnostics, and RF systems through high-level physics units rather than raw process variables (PVs) or equivalents.

At its core, CATAP utilizes procedural generation driven by YAML configuration files defined in a LAURA lattice. This ensures that the software remains facility-independent; by simply updating the configuration schema, the same physics scripts can be deployed across different accelerators without refactoring the logical flow. Jinja templating is used to develop the facility-specific base models in YAML, allowing rapid translation of existing controls information into CATAP.

pyCATAP employs the concept of 'factories' to group similar hardware objects into a single class (see Fig. 3). A magnet factory, for instance, could group all facility or machine-area magnets into a single object, which can then be acted upon as a set to, say, turn `ON` all magnets, or `DEGAUSS` all magnets. High-level systems, such as a variable bunch compressor or a photoinjector laser system, can also be instantiated. These systems combine disparate hardware objects, such as magnets and motion controls, or mirrors, energy meters and shutters into a single object for coordinated operations.

pyCATAP can use controls information from the LAURA YAML descriptions, which automatically allows a controls system interface to be generated from the ontological description in LAURA. Since pyCATAP is generated directly from these definitions, it ensures that any code used to control the accelerator is synchronised with the changes to the lattice, without the need to completely redefine any subsequent code.

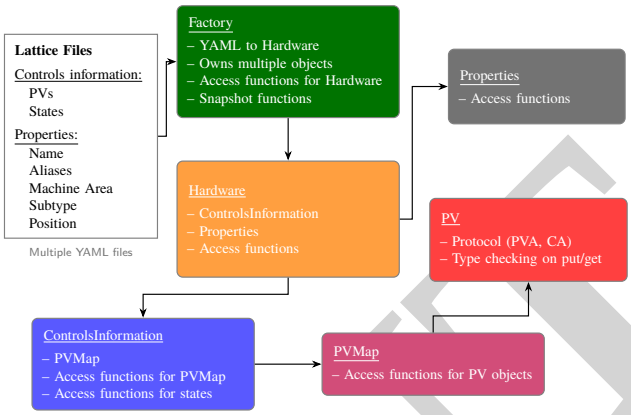

Figure 3: Class architecture of pyCATAP. A `Factory` reads LAURA YAML lattice files to instantiate `Hardware` objects, each composed of nested Pydantic models: `ControlsInformation` (EPICS PV access and device states via a `PVMap`) and `Properties` (physical metadata).

PUMBA: Experimental Control and Automation

PUMBA is a Python package for automating and standardising multi-step experimental procedures, built on top of pyCATAP and designed to interact directly with LAURA. Its purpose is to take the hardware access provided by pyCATAP and the lattice context provided by LAURA and encode them into reproducible, well-structured operational tasks that can be executed consistently.

The central design unit of PUMBA is the Procedure: a base class that enforces a strict lifecycle on every operational task. Every procedure must implement three phases:

- **Prepare**: performs all setup actions and validates that preconditions are met;
- **Run**: executes the core task
- **Clean-up**: restores any transient machine state and finalises outputs.

Procedures carry metadata including a name, a creation timestamp, a working directory for output files, and a logging instance, ensuring that each run is fully traceable.

PUMBA defines several specialised subclasses of Procedure:

- A **HealthCheck** is a read-only assessment of machine state: it interrogates hardware objects via pyCATAP and emits structured warnings if any element is not in its expected condition.
- A **ScanProcedure** is a stepped measurement in which a set of pre-defined machine set-points is applied in sequence — incrementally adjusting one or more hardware parameters — and data is recorded at each step. The internal adjust() and record() hooks define what is

changed and what is measured. The framework handles progress tracking, logging, and error recovery.

- A **SetupProcedure** is used to configure hardware to a target state without performing a measurement.

All three subclasses are composable into a Sequence, which executes a list of procedures in strict order, with optional early termination if any step fails. Sequences can be defined programmatically or loaded from YAML definition files. This is relevant for complex commissioning tasks: the steps of a multi-hour machine setup can be encoded as a YAML sequence, reviewed and approved offline, and then executed reproducibly.

Data Model: PUMBA relies on the use of a standardised data model for all experimental inputs and outputs. Recorded data is wrapped in a typed `DataValue` object, which carries a `DataContext`: a record of the lattice element where the data was measured, the machine area, and the precise timestamp. Element names are validated via LAURA. Specialised subclasses of `DataValue` exist for each hardware type — `MagnetData`, `CameraData`, `ChargeData`, etc. — carrying the relevant physical quantities with appropriate units and validation. This structured approach ensures that data from different procedures can be combined, compared, and archived in a consistent form.

Snapshots: PUMBA provides machine snapshot functionality, recording the instantaneous state of every hardware subsystem as a single timestamped object. Snapshots are taken automatically by procedures at configurable points in their lifecycle and stored alongside measurement data, providing a complete machine context for post-experiment analysis.

Alerts: An alert system allows procedures to register Process Variable (PV) monitors that run in the background during execution and abort or warn if nominated signals move outside safe bounds — e.g. monitoring RF system trips during long beam measurements.

Example procedures implemented within PUMBA currently include quadrupole scans for emittance reconstruction, magnet degaussing sequences, machine image "snapshotting", and optics matching workflows. These are operational tasks performed routinely on CLARA, implemented as reproducible, documented, software objects. Integrating PUMBA with SIMBA allows for a simulation-first workflow — using SIMBA to predict the expected beam response — before being executed on the real machine.

SARABI: Generating IOCs from LAURA

SARABI is a code-generation tool that produces soft EPICS IOCs from LAURA lattices. Its provides the simulation layer with a virtual control system: physics results computed by SIMBA are written to SARABI-generated soft PVs, making them indistinguishable from real hardware read-backs to any client on the network.

Each device in the LAURA lattice is described by a Python object with the same controls-information schema used by pyCATAP, recording every PV handle together with its signal type (`scalar`, `statistical`, `state`, `binary`, `waveform`), engineering units, and its EPICS protocol (Channel Access (CA) or PV Access (PVA)). A `SchemaTranslator` class maps field names between naming conventions; translations for LAURA keywords such as `controls_information` and `identifier` can be provided, allowing SARABI to consume device definitions written for non-LAURA facilities without modification.

At render time, the package walks the device YAML directory, builds a `PVInfo` object per device, and separates its PV map into common PVs (present on every device of that type) and unique PVs (device-specific). Jinja2 templates consume these maps to emit Python `PVGroup` classes using `caproto` for CA and `p4p` for PVA. A companion script generates an entry point per device type that instantiates and runs all IOCs concurrently. All generated PV names carry a `VM-` prefix to prevent any accidental overlap with physical control-system PVs on the same network.

Standard EPICS command-line tools can be used to read or write the virtual PVs once the appropriate environment variables are set.

THE INTEGRATED MODEL: A PROTOTYPE DIGITAL SHADOW

The integrated model for CLARA is implemented as a microservices architecture, deployed as a collection of coordinated Docker containers. The system connects a live (or virtual) EPICS control system to a running instance of SIMBA through a chain of middleware services, with simulation results published back onto the EPICS network as simulation PVs. This allows downstream applications to consume beam parameters from the simulation in an analogous manner to data from hardware diagnostics.

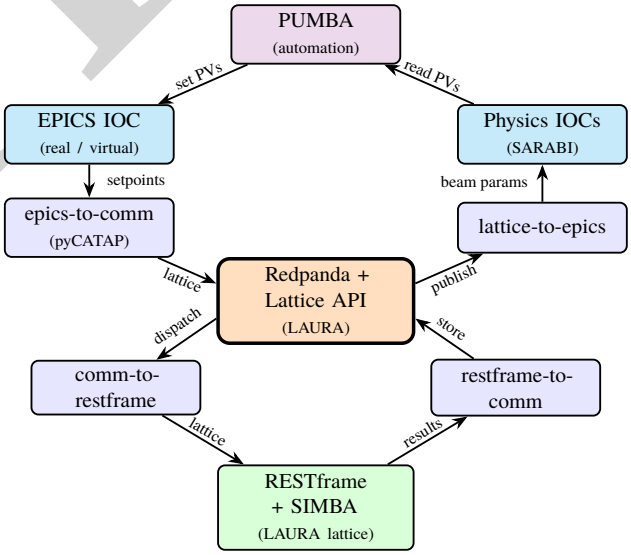

Figure 4: Data-flow loop of the CLARA integrated model. Additional control modules for facilitating data flow between codes are also shown.

The communication backbone of the system is a message broker based on Redpanda, a Kafka-compatible streaming

platform. Redpanda mediates the flow of lattice state between all services, decoupling producers (services that emit a state change, like a simulation completing) from consumers (services that act based on state changes, like sending results to virtual PVs).

The Data Flow

The system operates through the following sequence (Fig. 4), which is driven by machine changes (via PUMBA).

1. **Hardware state ingestion.** `epics-to-comms` monitors EPICS using pyCATAP, reading hardware setpoints (magnets, RF, etc). Changes to hardware states are translated to a minimal LAURA object and pushed to the central API.
2. **Lattice storage and versioning.** The API keeps track of settings in memory and stores completed simulation results in a database, creating a history of every simulated machine state.
3. **Simulation dispatch.** `comm-to-restframe` receives messages from `Kafka` when the Lattice API receives a new lattice. This is passed to the Simulation API – wrapping a SIMBA instance – which applies the LAURA model and begins tracking.
4. **Tracking and result generation.** SIMBA tracks the lattice using the specified simulation codes for each section. Twiss parameters and full phase-space distributions are computed along with synthetic screen images. *RESTFrame ensures only modified sections (and later) are re-tracked.*
5. **Result retrieval and storage.** `restframe-to-comms` receives messages from `Kafka` when tracking is complete, retrieves the output lattice (populated with simulation results), and posts it back to the Lattice API under the same UUID, where it is stored into the database.
6. **Publication to EPICS.** `lattice-to-epics` receives a `Kafka` message when a new database entry occurs and requests that lattice via UUID and maps computed beam parameters to virtual PVs before sending them to the SARABI instance.
7. **PUMBA / Operator display.** A React-based web frontend connects to the system via a PV Web-socket bridge for interaction via web-interface. It uses the Lattice API for a browser-based view of the lattice state, simulation results, and beam parameter history. Automated procedures can be performed via PUMBA.

Current Capabilities

The stack is functional end-to-end for virtual CLARA, deployable from a single `docker compose` command. An instance has also been tested with a model of the ISIS medium-energy injector (MEBT). The current implementations directly shadow the accelerator and the control system, but translation from physical quantities to simulation still needs refinement. The biggest hurdle towards full digital shadowing is simulation latency - current round trip times are on the order of seconds, rather than the µs required for true shadowing.

Replacing computationally expensive simulations with machine learning surrogate models has been implemented using the Poly-Lithic library [14], and tested for the ISIS neutron source MEBT section. A PyTorch model was trained using ASTRA simulations, and the model inputs and outputs matched to the element definitions in the LAURA lattice. With this structured format, SIMBA is able to track a given machine section in Poly-Lithic, treating it the same way as a conventional simulation code. Toggling the simulation code for a given machine section (via the virtual control system), allows surrogate models to greatly expedite the update loop, providing a realistic path towards a real-time digital twin.

FUTURE WORK

As noted, the Integrated Model acts a version '0.2' digital shadow for CLARA, with near term targets including further integration with Poly-Lithic and improved alignment between the model physics and the real machine. The upcoming CLARA beam commissioning and machine development period will help in this regard.

The long term goals of this project are two-fold:

1. **Extend the digital twin to new facilities (existing and planned)**: Simplify the creation of LAURA lattices; Create a robust shadow from only a LAURA lattice; Redefine 'start' and 'end' to encompass more physics (photo-injector lasers, experimental beam-lines etc).
2. **Design and optimise a new facility entirely within the shadow/twin**: Integrate xopt [15]/Badger [16] into the shadow; design and implement a representative experiment using PUMBA; optimise the LAURA lattice using PUMBA;

As part of the development process we expect to release a self-contained digital twin as a Docker image in the coming months.

CONCLUSION

A prototype digital shadow has been created bringing together five different codebases: an ontological accelerator description language (LAURA), a start-to-end simulation framework (SIMBA), an experimental automation package (PUMBA), a controls abstraction layer (pyCATAP), and a virtual EPICS IOC controller (SARABI). Using Redpanda/Kafka streaming and simple endpoint interfaces, a complete data loop has been created for two different scenarios – the CLARA linear electron accelerator and the ISIS proton injector beamline. The implementation of surrogate models has been proven with the ISIS model, and expansion of these models to the CLARA facility is expected to happen in the coming months. This prototype digital shadow demonstrates a viable ecosystem for development of accelerator digital shadows and represents a stepping-stone towards a more cohesive digital twin project.

REFERENCES

[1] D. Angal-Kalinin *et al*, "Design, specifications, and first beam measurements of the compact linear accelerator for research and applications front end", *Phys. Rev. Accel. Beams*, vol. 23, p. 044801, 2020. doi:10.1103/PhysRevAccelBeams.23.044801

[2] A. D. Brynes, J. K. Jones, M. King, *et al*, "A flexible start-to-end simulation framework for particle accelerators based on a comprehensive lattice description", *arXiv preprint*, 2026. doi:10.48550/arXiv.2604.19103

[3] M. King, A. D. Brynes, F. Jackson, *et al*, "Controls abstraction towards accelerator physics: A middle layer python package for particle accelerator Ccntrol", *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.19794

[4] A. D. Brynes, M. King, K. R. L. Baker *et al.*, "Closing the loop: Deploying auto-generating digital twins for particle accelerators", *arXiv preprint*, 2026. doi:10.48550/arXiv.2604.19101

[5] M. Borland, "ELEGANT: A flexible SDDS-compliant code for accelerator simulation", Advanced Photon Source, ANL, Lemont, IL, USA, Rep. LS-287, Sep. 2000.

[6] S. M. Lidia, K. Floettmann, and P. Piot, "Recent improvements to the ASTRA particle tracking code", in *Proc. PAC'03*, Portland, OR, USA, May 2003, paper FPAG015, pp. 3500–3502.

[7] M. J. de Loos and S. B. van der Geer, "General Particle Tracer: A new 3D code for accelerator and beamline design," in *Proc. 5th Eur. Part. Acc. Conf. (EPAC'96)*, Sitges, Spain, 1996, pp. 1241–1243.

[8] O. Stein, I.V. Agapov, A. Eichler, and J. Kaiser, "Accelerating Linear Beam Dynamics Simulations for Machine Learning Applications", *in Proc. IPAC'22*, Bangkok, Thailand, Jun. 2022, pp. 2330-2333. doi:10.18429/JACoW-IPAC2022-WEPOMS036

[9] J. Kaiser, C. Xu, A. Eichler, and A. Santamaria Garcia, "Bridging the gap between machine learning and particle accelerator physics with high-speed, differentiable simulations", *Phys. Rev. Accel. Beams*, vol. 27, p. 054601, 2024. doi:10.1103/PhysRevAccelBeams.27.054601

[10] I. Agapov, G. Geloni, S. Tomin, and I. Zagorodnov, "OCELOT: a software framework for synchrotron light source and FEL studies", *Nucl. Instrum. Methods Phys. Res. A*, vol. 768, pp. 151–156, 2014. doi:10.1016/j.nima.2014.09.057

[11] A. Adelmann *et al.*, "OPAL a versatile tool for charged particle accelerator simulations", *arXiv preprint*, 2019. doi:10.48550/arXiv.1905.06654

[12] A. Ferran Pousa, R. Assmann, and A. Martinez de la Ossa, "Wake-T: a fast particle tracking code for plasma-based accelerators", *J. Phys.: Conf. Ser.*, vol. 1350, p. 012056, 2019. doi:10.1088/1742-6596/1350/1/012056

[13] S. Reiche, "GENESIS 1.3: a fully 3D time-dependent FEL simulation code.", *Nucl. Instrum. Methods Phys. Res. A*, vol. 429, pp. 243–248, 1999. doi:10.1016/S0168-9002(99)00114-X

[14] M. Leputa, K. R. L. Baker, M. Romanovschi, and R. Banerjee, "Machine Learning for ISIS Controls", in *Proc, ICALEPCS'25*, Chicago, IL, USA, pp. 1127–1232, 2025. doi:10.18429/JACoW-ICALEPCS2025-WEPD078

[15] R. Roussel, C. Mayes, A. Edelen, and A. Bartnik, "Xopt: A simplified framework for optimization of accelerator problems using advanced algorithms", in *Proc. IPAC'23*, Venice, Italy, May 2023, pp. 4847–4850. doi:10.18429/JACoW-IPAC2023-THPL164

[16] R. Roussel, D. Kennedy, A. Edelen, N. Kuklev, S. Miskovich, and Z. Zhang, "Xopt and Badger: a machine learning ecosystem for real-time accelerator control and optimization", in *Proc. ICALEPCS'25*, Chicago, IL, USA, Sep. 2025, pp. 1271–1275. doi:10.18429/JACoW-ICALEPCS2025-WESV001