The Experiment Orchestration System (EOS): Comprehensive Foundation for Laboratory Automation

Angelos Angelopoulos[1], Cem Baykal[1], Jade Kandel[1], Matthew Verber[2], James F. Cahoon[2], Ron Alterovitz[1]

***Abstract*— As scientific research in chemistry, materials science, and applied sciences becomes increasingly complex and data-driven, there is a growing need for efficient, scalable, and flexible automation to accelerate discoveries and reduce human burden and error in laboratories. We introduce the Experiment Orchestration System (EOS), an open-source software framework and runtime offering a comprehensive foundation for laboratory automation. EOS offers an extensible framework allowing users to define labs, devices, tasks, experiments, and optimization criteria using YAML and Python plugins, and also offers a distributed runtime for managing and executing automation. EOS has a central orchestrator that communicates with and controls laboratory equipment to execute tasks. EOS implements autonomous experiment campaigns, parameter optimization, task scheduling, result aggregation, and more. By providing a common infrastructure for laboratory automation, EOS aims to reduce automation implementation barriers and accelerate discoveries in science laboratories.**

I. Introduction

Science laboratories in fields such as chemistry and materials science drive innovation in applications such as energy, sustainability, and electronics. This research often requires synthesizing, characterizing, and optimizing novel molecules and materials. Creating and optimizing molecules and materials requires navigating a vast chemical space of possible reagents, reactions, chemical systems, and process parameters. While intuition, knowledge, and computational modeling can help narrow the space, real-world experimentation is necessary to explore the chemical space and discover optimal solutions.

Robots and autonomous systems can automate tasks in science laboratories, producing experimental results faster and with greater reproducibility [1]. Automated laboratory tasks can include transporting samples, loading apparatuses, conducting measurements and analyses, and processing data. Successful implementations employ liquid handling robots that precisely dispense microliters of reagents [2], [3], robotic arms that transfer samples between adjacent apparatuses [4], [5], mobile manipulation robots that transport samples between spatially distant apparatuses [6], [7], and machine learning models that optimize experimental parameters [8].

*This work was supported by the Creativity Hub at the University of North Carolina at Chapel Hill and the University of North Carolina System Research Opportunities Initiative (ROI).

[1]Angelos Angelopoulos, Cem Baykal, Jade Kandel, and Ron Alterovitz are with the Department of Computer Science, University of North Carolina at Chapel Hill, NC 27599, USA {aangelos,kandelj,ron}@cs.unc.edu

[2]Matthew Verber and James F. Cahoon are with the Department of Chemistry, University of North Carolina at Chapel Hill, NC 27599, USA {mverber,jfcahoon}@unc.edu

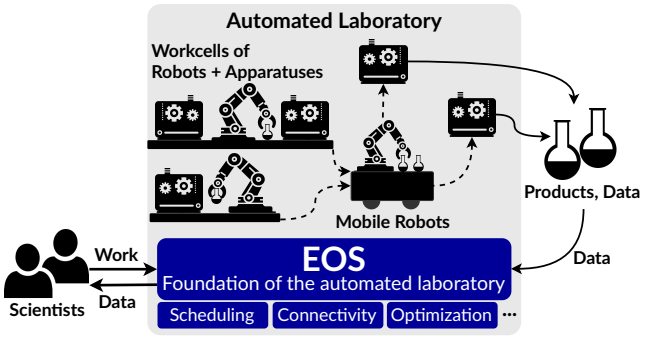

Fig. 1. EOS is a comprehensive foundation for building, managing, and running automated laboratories.

Automation can accelerate the search of vast chemical spaces and speed up scientific discovery.

Automating science laboratories is challenging. Diverse tasks that are constantly changing must be performed, specialized lab equipment may be shared between autonomous systems and scientists, and safety requirements must be satisfied [1]. These considerations make science lab automation ill suited to many manufacturing automation approaches, which work best when the same tasks are performed repeatedly. To efficiently automate laboratory tasks, we need software to coordinate and operate disparate robots and apparatuses to run experiments, adapt to new experiments and changes in the laboratory, collect and organize data, and integrate artificial intelligence (AI) for parameter optimization. The current lack of software infrastructure for lab automation hinders automation implementation, experiment protocol sharing and reproducibility, and automation reuse.

We introduce the Experiment Orchestration System (EOS), an open-source [9] framework and runtime (execution engine) offering a comprehensive foundation for laboratory automation. Using EOS, scientists can implement tasks and devices, define experiments, specify objectives, and EOS will schedule and execute the tasks, organize results, and optimize experiment parameters based on the objectives (Fig. 1). EOS can support any level of laboratory automation [1], from level 1 "assistive automation" in which EOS controls a single instrument to level 5 "full automation" in which EOS orchestrates entire multi-step experiments.

EOS enables scientists to set up automation and define their tasks and objectives using YAML [10] and Python. Scientists can then use EOS to run the automation. EOS orchestrates distributed tasks, communicating with and controlling laboratory apparatuses. Through experiment campaign management, task scheduling, automatic result aggregation,

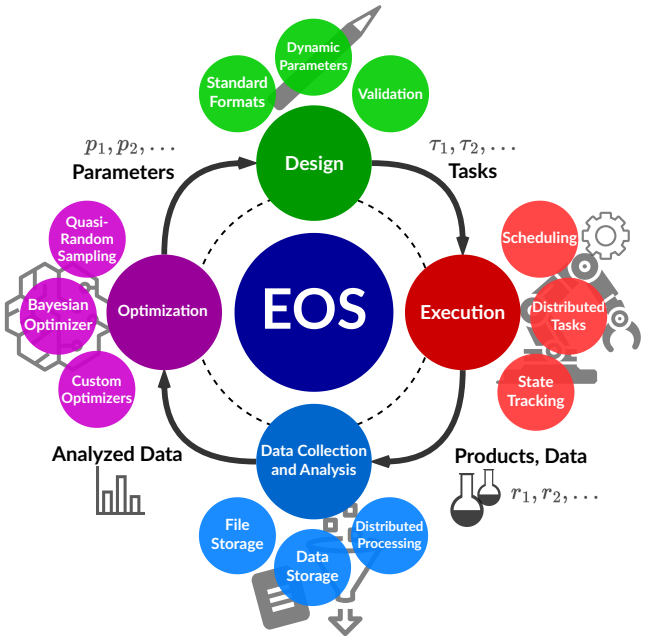

Fig. 2. EOS offers the foundation for laboratory automation.

parameter optimization, and validation, EOS enables closed-loop automation (Fig. 2). By lowering the barrier to implementing laboratory automation, EOS can greatly benefit both newcomers and existing laboratory automation users.

II. Related Work

There has been much work of varying complexity in laboratory automation. Implementations commonly used benchtop workcells [4], [5], [11], [12], [13], [14] consisting of apparatuses connected with fixed robot arms for batch processing or tubes for flow chemistry. Others have used Cartesian robots such as liquid handlers for rapid synthesis with multi-well plates [2], [15], dual-arm manipulators for faster task completion or bimanual dexterity [16], and even mobile robots for transporting samples and loading/unloading them to/from apparatuses [6], [17], [18], [7]. We briefly highlight two laboratory automation implementations [4], [6], which we selected due to their complexity.

Burger et al. [6] focused on photocatalysis, automating experiments to find active photocatalyst mixtures. A KUKA KMR iiwa [19] mobile manipulator transported samples between 9 stations containing apparatuses for tasks like dispensing, capping, sonication, and chromatography. The robot loaded/unloaded samples and operated equipment via API calls. Each experiment took 8 hours, and experiments were autonomously optimized. A custom "process management system" controlled the lab apparatuses and the robot through protocols such as TCP via Ethernet/Wi-Fi and RS-232.

Szymanski et al. [4] built an autonomous lab for inorganic powder synthesis. It comprised three workcells, each with a robot arm and apparatuses for chemistry. The lab operated for 17 days, conducting 355 experiments. Synthesis recipes were generated using AI models mining literature data. Temperatures were also predicted using AI and reactions were optimized using a custom method. The lab was controlled by "AlabOS". Equipment drivers and tasks were implemented in Python, with tasks executed as actors. The system tracked equipment and samples to prevent conflicts. Experiments were represented as directed acyclic graphs and executed based on equipment availability and prioritization.

Laboratory automation faces significant challenges due to diverse equipment and complex integration needs [1]. Research groups invest much time in developing software for device drivers, communication protocols, data pipelines, parameter optimization, and optimizing throughput, leading to redundant efforts with limited reusability. While systems like SiLA 2 [20] and HELAO-async [21] have attempted to address some of these challenges, their adoption has been limited. SiLA 2 is a client-server architecture for integrating laboratory automation components. Systems like SiLA 2 Manager [22] have added orchestration capabilities to SiLA 2, but they are more limited than EOS, lacking features such as optimized scheduling, campaign optimization, and user code hot reloading.

To address these limitations, we developed EOS as both a comprehensive framework and orchestration engine for laboratory automation. EOS combines ease of use with extensive functionality, including integration with external systems, support for diverse workflows, and features for autonomous experimentation. Unlike existing systems, EOS provides a comprehensive suite of capabilities including actor-based distributed communication, task scheduling, data aggregation, and parameter optimization. EOS abstracts complexity, is extensible, and is built to support the future of laboratory automation.

III. The Experiment Orchestration System

EOS supports crucial aspects needed for laboratory automation, illustrated in Fig. 2. During the *design* phase, rather than having to build complex systems from scratch, EOS provides a structured framework for defining laboratories, devices, tasks, and experiments as well as a package system for organizing them. For example, EOS offers Python and YAML interfaces for implementing device drivers and tasks. The framework provided by EOS saves scientists time and effort and makes automation easier to develop, understand, and share.

After the design phase, laboratory automation goes through the *execution*, *data collection and analysis*, and *optimization* phases. EOS is a comprehensive runtime for laboratory automation, handling aspects such as state management, data persistence, task scheduling, validation, parameter optimization, and distributed execution across lab computers. EOS also simplifies administrative tasks like updating code across laboratory computers through hot reloading and automatic code deployment. The major features of EOS are summarized in Fig. 3. In the following subsections we describe how EOS supports the key phases of lab automation, as shown in Fig. 2.

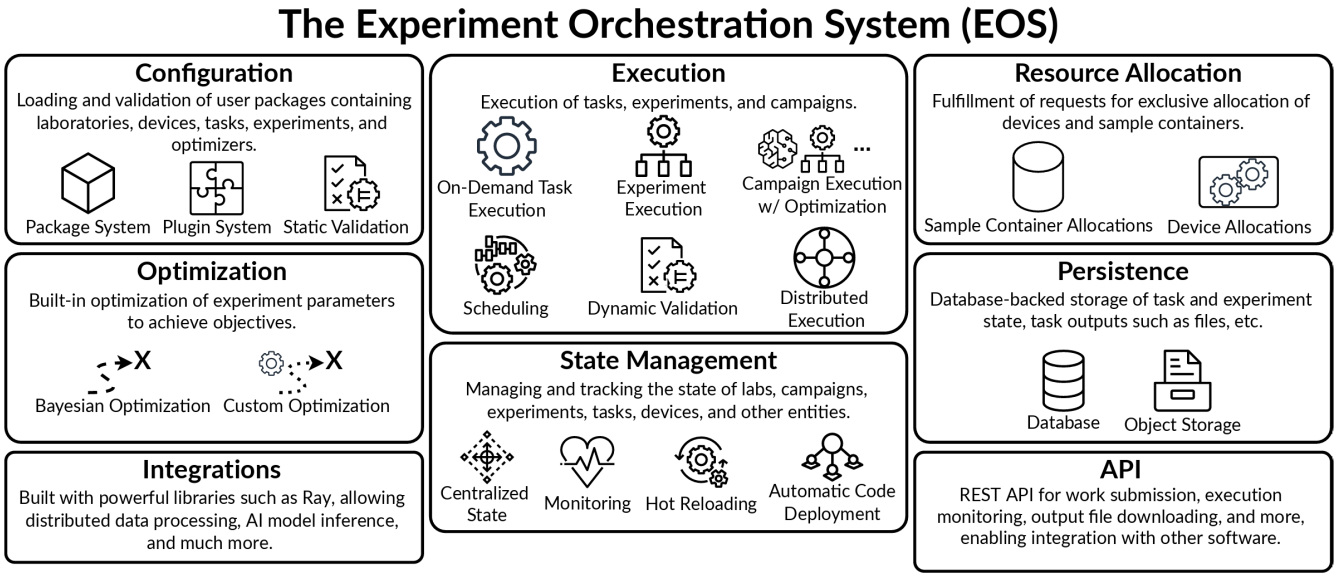

Fig. 3. The major features of EOS.

A. Design

During the design phase, EOS provides a structured framework for scientists to implement laboratory automation and design experiments. Here we present the key components that make up this framework.

Packages: Automation in EOS is organized in packages. A package can contain any combination of laboratories, experiments, devices, and tasks, along with arbitrary code and data. For instance, a package might contain task and device implementations for equipment from a specific manufacturer, while another might contain experiments designed for a particular lab. EOS packages can be shared with others, similarly to ROS [23] packages.

Laboratories: Laboratories house an array of components including computers, devices (e.g., apparatuses), and sample containers, all used for experiments. Scientists define each laboratory in a YAML file specifying (1) the locations in the lab (useful for mobile robot navigation), (2) the computers connected to EOS that can run work, (3) devices in the lab, with their specific locations and connected computers, and (4) sample containers, grouped by type (e.g., beaker, vial) and given unique identifiers.

Devices: Automation is executed by devices, such as laboratory apparatuses and robots, which require drivers enabling EOS to interface with the hardware and execute automation. In EOS, scientists define devices with two files: (1) a YAML file containing a unique "type" for the device (e.g., magnetic mixer) and any initialization parameters (e.g., socket ports for physical device connections), and (2) a Python file with the "driver" implementation.

Tasks: Tasks coordinate and control devices. Scientists define tasks using a YAML file and a Python file. The YAML file is an interface contract and documentation, specifying requirements such as the required device types, input/output parameters with data types, constraints, and containers. The Python file implements the task logic. Example YAML and Python files for a magnetic mixing task are shown in Listing 1 and Listing 2, respectively.

```yaml
type: Magnetic Mixing
desc: Mix the contents of a beaker

device_types:
  - magnetic_mixer

input_containers:
  beaker:
    type: beaker

input_parameters:
  mixing_time:
    type: int
    unit: sec
    min: 5
    desc: How long to mix

  mixing_speed:
    type: int
    unit: rpm
    min: 60
    max: 2000
    desc: How fast to mix
```

Listing 1. Sample YAML specification for a magnetic mixing task.

Experiments: An experiment is defined by a directed acyclic graph (DAG) of tasks. Scientists define each experiment in a YAML file containing (1) the laboratories involved, (2) the parameters to initialize containers, and (3) the tasks to be executed. Parameters can be hard-coded values or references to other task inputs and outputs. Task input parameters can also be dynamic, denoted as "eos_dynamic", which allows input to be provided at a later time by a scientist or by an optimizer, the latter enabling closed-loop experimentation. Tasks have dependencies to establish ordering. An excerpt of an experiment task sequence is shown in Listing 3.

Campaigns: A campaign is a sequence of repeated experiments whose parameters are provided by an optimizer. The goal of the campaign is to achieve some objectives by finding the optimal parameters. For example, the goal of a campaign

```python
class MagneticMixing(BaseTask):
  def _execute(
    self,
    devices: BaseTask.DevicesType,
    parameters: BaseTask.ParametersType,
    containers: BaseTask.ContainersType,
  ) -> BaseTask.OutputType:
    magnetic_mixer = devices.get_all_by_type(
      "magnetic_mixer"
    )[0]

    containers["beaker"] = magnetic_mixer.mix(
      containers["beaker"],
      parameters["mixing_time"],
      parameters["mixing_speed"]
    )

    return None, containers, None
```

Listing 2. Sample Python implementation for a magnetic mixing task.

```yaml
tasks:
  ...
  - id: dispense_colors
    type: Dispense Colors
    desc: Dispense colors into a container
    devices:
      - lab_id: color_lab
        id: color_dispenser
    containers:
      beaker: retrieve_container.beaker
    parameters:
      cyan_volume: eos_dynamic
      magenta_volume: eos_dynamic
      yellow_volume: eos_dynamic
      black_volume: eos_dynamic
    dependencies: [retrieve_container]

  - id: move_container_to_mixer
    type: Move Container
    desc: Move the container to the mixer
    devices:
      - lab_id: color_lab
        id: robot_arm
      - lab_id: color_lab
        id: magnetic_mixer
    containers:
      beaker: dispense_colors.beaker
    parameters:
      target_location: magnetic_mixer
    dependencies: [dispense_colors]
    ...
```

Listing 3. Excerpt from a YAML color mixing experiment definition.

could be to identify more efficient catalysts. Scientists can manually set dynamic parameters for experiments or delegate parameter selection entirely to the optimizer.

Validation: Scientists may make mistakes when implementing laboratories, devices, tasks, and experiments. It is important to catch as many mistakes as possible to increase reliability and safety. EOS performs static validation, checking for complete task parameters in experiments, unit specification for numeric parameters, device-computer correspondence, unique sample container IDs, and more. Runtime validations occur during task execution, such as ensuring parameters match task specifications in terms of data types and value constraints.

B. Execution

During the execution phase, laboratory automation is executed and results are generated. EOS supports this phase with its runtime (i.e., execution engine). After implementing automation in the EOS framework, the EOS runtime can manage and execute the automation.

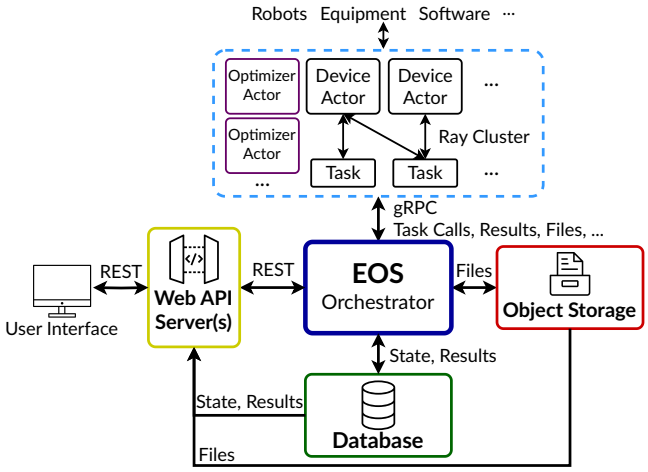

Fig. 4. The high-level system architecture of EOS.

The EOS runtime consists of a central orchestrator and a Ray [24] cluster. Ray is a framework and set of libraries for distributed computing. EOS connects to lab computers running Ray workers and creates persistent actor processes to control devices. EOS has a hierarchical execution model supporting tasks, experiments, and campaigns. The task executor is at the lowest level, and executes tasks submitted by users and higher-level executors. Experiment executors submit execution requests to the task executor for each task in the experiments. A scheduler coordinates experiments to prevent conflicts in device and container usage and improve throughput. At the top level, campaign executors create experiment executors for each campaign experiment. EOS's architecture is summarized in Fig. 4 and major features are shown in Fig. 3.

EOS loads tasks and devices as plugins at runtime. EOS spawns each device implementation as a persistent Ray actor process on the computer connected to the device, providing an interface between EOS and the device. Tasks in EOS are transient functions that call device functions through Ray. Tasks can be simple, using a single device for a single distinct process, or they can be more complex, grouping together processes by using multiple devices. EOS automatically deploys task and device code to the appropriate computers. For example, if scientists make changes to device code while EOS is running, the device can be reloaded and EOS will make a new Ray actor without restarting. The tasks and devices execution model is illustrated in Fig. 5.

The scheduler enables EOS to execute experiments while avoiding conflicts in shared devices and sample containers. The default scheduler uses a greedy policy, scheduling tasks based on experiment priorities and whenever resources are available. The scheduling system is modular, allowing for the integration of new schedulers in the future such as a scheduler that takes the charge level of a mobile robot into account.

Devices and sample containers can only be used by one task at a time. In addition, humans may need to use some devices or sample containers so EOS should not use them

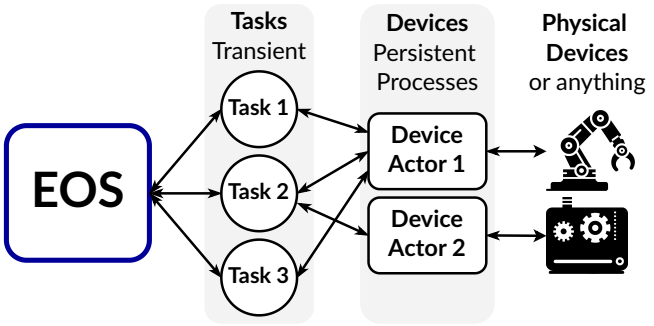

Fig. 5. The execution model using tasks and devices in EOS.

for the duration. To ensure safety, EOS allocates devices and sample containers to "owners" such as tasks or humans. The scheduler submits resource allocation requests for experiment tasks and the task executor for standalone tasks. Requests have a priority and a timeout after which they must be re-issued to avoid deadlocks. Tasks can execute only when their requested resources have been allocated.

Finally, managing state is crucial. Here, "state" refers to the current condition and information of all system components. To prevent state corruption that may arise from bad practices or bugs in user code, EOS centralizes state. The EOS orchestrator manages system state such as task and experiment statuses as well as creates and destroys Ray actors for devices and campaign optimizers. EOS also resolves parameter and container references used as inputs to tasks. Tasks and devices do not directly edit state as they do not have access to the EOS database.

C. Data Collection and Analysis

During the data collection and analysis phase, EOS aggregates data generated by tasks and experiments. EOS stores the data and offers features to help with data processing.

EOS automatically stores all state data for tasks, experiments, and campaigns in a database (PostgreSQL [25]) and an object store (MinIO [26]). For example, EOS stores state for recovering from disruptions (e.g., experiment progress), as well as task results and output files. Persistence is seamless and requires no user input. For example, EOS aggregates task results and stores files in the object store automatically. Users can easily access stored data to validate and replicate work later.

EOS also supports scalable data analysis through distributed computing. For instance, scientists can leverage Ray's data library for distributed processing of large data sets, enabling automated data analysis. Scientists can also use Ray to train machine learning and reinforcement learning models, serve AI models for inference by other tasks, or run any distributed tasks and actors they need. These features are available through EOS tasks and devices.

D. Optimization

In the final phase, EOS uses analyzed data to optimize future experiments. EOS includes a built-in Bayesian optimizer using BoFire [8]. Through BoFire, the built-in optimizer supports single and multi-objective optimization, surrogate models such as Gaussian processes and Bayesian neural networks, several acquisition functions, and quasi-random sampling such as Sobol and Latin Hypercube. Optimizers are defined as Python plugins, integrated within the corresponding experiment's EOS package. Scientists can use the built-in optimizer as is, extend it, or provide their own optimizers. Optimizers are automatically spawned as Ray actors on computers specified by the user.

IV. Example Scenario

In this section, we show how EOS can facilitate lab automation and optimize an objective. In the example, we have a laboratory for color mixing. The objective is to learn how to synthesize a target color specified using RGB (red, green, blue) with CMYK (cyan, magenta, yellow, black) ingredient colors while minimizing the used ingredients. The input space has 10 continuous parameters: a color volume (amount) and color intensity (dilution) for each CMYK component, and a mixing time and speed. The number of parameters and possible colors is in the millions, creating a large optimization domain. Although analytical color mixing models could accelerate this process, our goal is to demonstrate an intuitive end-to-end example of how a laboratory and optimization objectives can be set up in EOS, and how leveraging EOS's features can enable quicker lab automation. This example's source code and simulation is available to download as an EOS package [27].

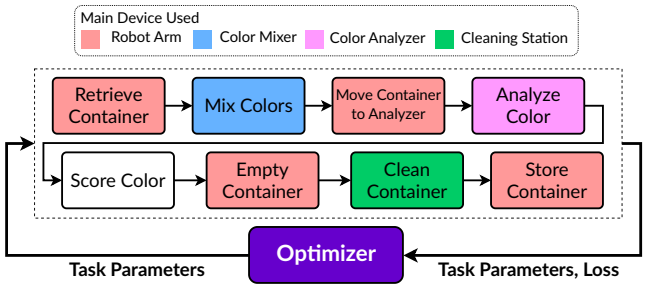

Fig. 6. The color mixing experiment. Most tasks use multiple devices, but we denote the most important device by color.

A. Design

We implemented an abstract virtual lab with 8 devices: a robot arm, 3 color mixers, 3 color analyzers, and a cleaning station. We defined the lab with EOS, implemented the code for the 4 unique devices, and implemented 8 tasks as shown in Fig. 6. The experiment begins with the "Retrieve Container" task, which grabs a specified container with the robot arm, and "moves" the container to the color mixer for the next task. We simulate container movement by updating their locations. The last task is "Store Container", which uses the robot arm to store the container used for the experiment.

B. Optimization, Data Collection, Analysis

The objective is to minimize the loss $\mathscr{L}(c,t) = w_1\sqrt{\sum_{i\in r,g,b}(c_i-t_i)^2} + w_2 v$, where c is the synthesized RGB

color, t is the target RGB color, c_i are the RGB components of c, t_i are the RGB components of t, v is the total dispensed color volume, and $w_1 = 0.8$ and $w_2 = 0.2$ are weights. The loss has two normalized components: (1) a Euclidean distance for the problem of synthesizing a target color, and (2) v to penalize large and redundant color volumes.

We configured EOS's built-in Bayesian optimizer to use the Upper Confidence Bound acquisition function, and configured 50 Sobol-sampled experiments to collect initial data. Subsequently, we conducted 100 experiments with optimization. We constrained color volume to $[0,25]$, color intensity to $[2,100]\%$, mixing time to $[1,45]$ seconds and mixing speed to $[100,200]$. Color volume and mixing time are unitless, and are specific to the fluid simulation we used to simulate color mixing, which we describe later in this section. The optimizer generates task parameters for each experiment by considering the task parameters and losses of past experiments (Fig. 6).

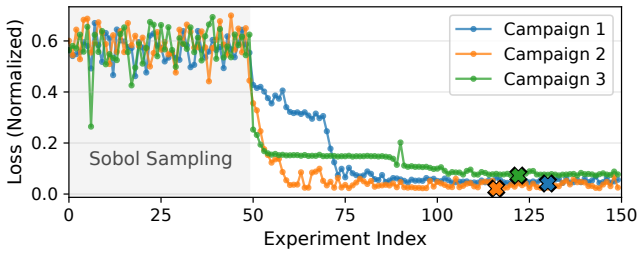

Fig. 7. Losses for the color mixing campaigns. The lowest losses are marked with "X". The campaigns progress concurrently; EOS handles scheduling, data collection, and optimization automatically.

TABLE I
RESULTS OF THE THREE COLOR MIXING CAMPAIGNS.

Campaign	Target RGB	Best RGB	Lowest Loss
1	(53, 29, 64)	(57, 28, 53)	0.04
2	(11, 25, 82)	(11, 26, 82)	0.02
3	(14, 107, 20)	(38, 99, 27)	0.07

C. Execution and Results

We ran three simultaneous campaigns each consisting of 150 experiments (totaling 450 experiments and 3600 task executions) with Bayesian optimization to discover how to synthesize three distinct colors specified by the user. To enable anyone to run this color mixing example with EOS, we simulate the devices and tasks in the virtual laboratory. Excluding mixing and robot arm movements, we configured each task to take 1 second. In order to move a container from one device to another, the robot arm requires multiple movements, each of which is configured to take 1 second. We assume that the robot arm is situated in the center of a table. For example, if a container needs to move from the color mixer to the color analyzer, then the arm has to do three movements: (1) from the center to the color mixer, (2) from the color mixer to the analyzer, and (3) back to the center.

The "Mix Colors" and "Analyze Color" tasks interface with a dedicated fluid simulation over a web socket, sending commands to dispense CMYK colors at different amounts and intensities, setting the mixing speed, and computing the average color. "Mix Colors" runs the simulation and is the longest task, configured to take up to 45 seconds. The color mixer device commands the simulation to dispense and mix the colors in reverse CMYK order. "Analyze Color" computes the average RGB color on the canvas and is used by the "Score Color" task to compute $\mathscr{L}$.

To simulate color mixing, we used a real-time GPU-accelerated fluid simulation [28], [29] based on Stable Fluids [30], and a physically-based color mixing model using Kubelka-Munk theory [29]. The fluid simulation runs using WebGL in a browser and uses shaders to apply non-linear operations over time (e.g., Navier-Stokes equations). For fluid mixing, we implemented a vortex with configurable speed. CMYK colors of different amount and intensity can be placed in the canvas. Color volume is simulated by changing the radius of the liquid color spawned as a circle in the simulation, while mixing speed is an amplitude multiplier that controls the speed of the vortex. We treat the simulation as a complex black box function, aiming to discover how to use it to synthesize target colors. Each campaign uses a dedicated fluid simulation, so there are three browser windows running concurrently.

EOS streamlined setting up the color mixing experiment by providing a framework for concise implementation of devices and tasks. We set up three experiments, one for each color target. We then set up the optimizers by defining the parameters, objective, initial sampling, and acquisition function. We submitted a campaign for each of the 3 experiments, and EOS took care of scheduling, task execution, and optimization. The scheduler ran the campaigns concurrently for about 2 hours, executing tasks and using devices in parallel, saving approximately 4 hours compared to running sequentially. The results are summarized in Fig. 7 and Table I. Overall, EOS greatly simplified the process of defining the color mixing problem, and made it easy to run experiments and generate results.

V. CONCLUSIONS

EOS is an open-source [9] framework and runtime for implementing and executing laboratory automation. It includes a comprehensive feature set for laboratory automation such as: YAML and Python plugins for implementing tasks, device interfaces, and experiments, a package system, static and dynamic validation, on-demand task, experiment, and campaign execution, scheduling, Bayesian optimization, result aggregation, and distributed execution. EOS was developed from our practical need for robust lab automation infrastructure that could integrate diverse equipment including robots, custom apparatuses, and AI systems. Moving forward, we plan many enhancements for EOS and hope it will benefit the scientific community by lowering barriers to entry in the rapidly growing field of laboratory automation.

References

[1] A. Angelopoulos, J. F. Cahoon, and R. Alterovitz, "Transforming science labs into automated factories of discovery," *Science Robotics*, vol. 9, no. 95, p. eadm6991, Oct. 2024, publisher: American Association for the Advancement of Science.

[2] F. Kong, L. Yuan, Y. F. Zheng, and W. Chen, "Automatic Liquid Handling for Life Science: A Critical Review of the Current State of the Art," *Journal of Laboratory Automation*, vol. 17, no. 3, pp. 169–185, June 2012, publisher: SAGE Publications Inc.

[3] P. Dettinger, T. Kull, G. Arekatla, N. Ahmed, Y. Zhang, F. Schneiter, A. Wehling, D. Schirmacher, S. Kawamura, D. Loeffler, and T. Schroeder, "Open-source personal pipetting robots with live-cell incubation and microscopy compatibility," *Nature Communications*, vol. 13, no. 1, p. 2999, May 2022, publisher: Nature Publishing Group.

[4] N. J. Szymanski, B. Rendy, Y. Fei, R. E. Kumar, T. He, D. Milsted, M. J. McDermott, M. Gallant, E. D. Cubuk, A. Merchant, H. Kim, A. Jain, C. J. Bartel, K. Persson, Y. Zeng, and G. Ceder, "An autonomous laboratory for the accelerated synthesis of novel materials," *Nature*, vol. 624, no. 7990, pp. 86–91, Dec. 2023, publisher: Nature Publishing Group.

[5] A. Sparkes, W. Aubrey, E. Byrne, A. Clare, M. N. Khan, M. Liakata, M. Markham, J. Rowland, L. N. Soldatova, K. E. Whelan, M. Young, and R. D. King, "Towards Robot Scientists for autonomous scientific discovery," *Automated Experimentation*, vol. 2, no. 1, p. 1, Jan. 2010.

[6] B. Burger, P. M. Maffettone, V. V. Gusev, C. M. Aitchison, Y. Bai, X. Wang, X. Li, B. M. Alston, B. Li, R. Clowes, N. Rankin, B. Harris, R. S. Sprick, and A. I. Cooper, "A mobile robotic chemist," *Nature*, vol. 583, no. 7815, pp. 237–241, July 2020, publisher: Nature Publishing Group.

[7] A. Angelopoulos, M. Verber, C. McKinney, J. Cahoon, and R. Alterovitz, "High-Accuracy Injection Using a Mobile Manipulation Robot for Chemistry Lab Automation," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2023, pp. 10 102–10 109, iSSN: 2153-0866.

[8] J. P. Dürholt, T. S. Asche, J. Kleinekorte, G. Mancino-Ball, B. Schiller, S. Sung, J. Keupp, A. Osburg, T. Boyne, R. Misener, R. Eldred, W. S. Costa, C. Kappatou, R. M. Lee, D. Linzner, D. Walz, N. Wulkow, and B. Shafei, "BoFire: Bayesian Optimization Framework Intended for Real Experiments," Aug. 2024, arXiv:2408.05040 [cs, math, stat].

[9] A. Angelopoulos, "UNC-Robotics/eos: The Experiment Orchestration System (EOS)." [Online]. Available: https://github.com/UNC-Robotics/eos

[10] "The Official YAML Web Site," Feb. 2025. [Online]. Available: https://yaml.org/

[11] J. M. Granda, L. Donina, V. Dragone, D.-L. Long, and L. Cronin, "Controlling an organic synthesis robot with machine learning to search for new reactivity," *Nature*, vol. 559, no. 7714, pp. 377–381, July 2018, number: 7714 Publisher: Nature Publishing Group.

[12] A.-C. Bédard, A. Adamo, K. C. Aroh, M. G. Russell, A. A. Bedermann, J. Torosian, B. Yue, K. F. Jensen, and T. F. Jamison, "Reconfigurable system for automated optimization of diverse chemical reactions," *Science*, vol. 361, no. 6408, pp. 1220–1225, Sept. 2018, publisher: American Association for the Advancement of Science.

[13] S. Steiner, J. Wolf, S. Glatzel, A. Andreou, J. M. Granda, G. Keenan, T. Hinkley, G. Aragon-Camarasa, P. J. Kitson, D. Angelone, and L. Cronin, "Organic synthesis in a modular robotic system driven by a chemical programming language," *Science*, vol. 363, no. 6423, p. eaav2211, Jan. 2019.

[14] C. W. Coley, D. A. Thomas, J. A. M. Lummiss, J. N. Jaworski, C. P. Breen, V. Schultz, T. Hart, J. S. Fishman, L. Rogers, H. Gao, R. W. Hicklin, P. P. Plehiers, J. Byington, J. S. Piotti, W. H. Green, A. J. Hart, T. F. Jamison, and K. F. Jensen, "A robotic platform for flow synthesis of organic compounds informed by AI planning," *Science*, vol. 365, no. 6453, p. eaax1566, Aug. 2019, publisher: American Association for the Advancement of Science.

[15] H. Tegally, J. E. San, J. Giandhari, and T. de Oliveira, "Unlocking the efficiency of genomics laboratories with robotic liquid-handling," *BMC Genomics*, vol. 21, no. 1, p. 729, Oct. 2020.

[16] Y. Jiang, H. Fakhruldeen, G. Pizzuto, L. Longley, A. He, T. Dai, R. Clowes, N. Rankin, and A. I. Cooper, "Autonomous biomimetic solid dispensing using a dual-arm robotic manipulator," *Digital Discovery*, vol. 2, no. 6, pp. 1733–1744, 2023, publisher: Royal Society of Chemistry.

[17] S. Kleine-Wechelmann, K. Bastiaanse, M. Freundel, and C. Becker-Asano, "Designing the mobile robot Kevin for a life science laboratory," in *2022 31st IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, Aug. 2022, pp. 870–875, iSSN: 1944-9437.

[18] H. Liu, N. Stoll, S. Junginger, and K. Thurow, "Mobile robotic transportation in laboratory automation: Multi-robot control, robot-door integration and robot-human interaction," in *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO 2014)*, Dec. 2014, pp. 1033–1038.

[19] "KMR iiwa." [Online]. Available: https://www.kuka.com/en-us/products/amr-autonomous-mobile-robotics/mobile-robot-systems/kmr-iiwa

[20] D. Juchli, "SiLA 2: The Next Generation Lab Automation Standard," in *Smart Biolabs of the Future*, ser. Advances in Biochemical Engineering/Biotechnology, S. Beutel and F. Lenk, Eds. Cham: Springer International Publishing, 2022, pp. 147–174.

[21] D. Guevarra, K. Kan, Y. Lai, R. J. R. Jones, L. Zhou, P. Donnelly, M. Richter, H. S. Stein, and J. M. Gregoire, "Orchestrating nimble experiments across interconnected labs," *Digital Discovery*, vol. 2, no. 6, pp. 1806–1812, 2023, publisher: Royal Society of Chemistry.

[22] L. Bromig, D. Leiter, A.-V. Mardale, N. von den Eichen, E. Bieringer, and D. Weuster-Botz, "The SiLA 2 Manager for rapid device integration and workflow automation," *SoftwareX*, vol. 17, p. 100991, Jan. 2022.

[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, and others, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3. Kobe, Japan, 2009, p. 5, issue: 3.2.

[24] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," Sept. 2018, arXiv:1712.05889 [cs, stat].

[25] PostgreSQL Global Development Group, "PostgreSQL." [Online]. Available: https://www.postgresql.org/

[26] MinIO Inc, "MinIO | S3 Compatible Storage for AI." [Online]. Available: https://min.io

[27] A. Angelopoulos, "UNC-Robotics/eos-examples: Collection of example EOS packages." [Online]. Available: https://github.com/UNC-Robotics/eos-examples

[28] P. Dobryakov, "PavelDoGreat/WebGL-Fluid-Simulation," Sept. 2024, original-date: 2017-08-22T21:30:33Z. [Online]. Available: https://github.com/PavelDoGreat/WebGL-Fluid-Simulation

[29] "Mixbox - Natural Color Mixing Based on Real Pigments." [Online]. Available: https://scrtwpns.com/mixbox

[30] J. Stam, "Stable fluids," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. USA: ACM Press/Addison-Wesley Publishing Co., July 1999, pp. 121–128.