1 Introduction

Attempting to do something for the first time is always a big undertaking. Whether this "something" is performed by following in the footsteps of others or by forging one’s own path toward a working method, a systematic approach can be of great help. Trying a new method a number of times and with varying approaches, one should be able to reach a method encapsulating each step, which makes sense and is manageable to perform repeatedly.

In having discovered and implemented this method, one has found a workflow for the process. This workflow will contain both the order and nature of the actions to perform and the resources, both time and materials, required for each. By examining the overall workflow, one can split it into discrete tasks, each of which might show up multiple times throughout the workflow. These tasks can require different resources but be the same process, e.g., measuring an amount of a material, mixing chemicals in aqueous media, performing a computational single-point calculation, or training a machine learning (ML) model. Tasks can depend on the completion of other tasks, and they can run in parallel with each other. A collection of tasks can also be run over and over until some criterion is satisfied. Of course, it is important that each of these tasks is done correctly, but in the context of workflows, the input, output, required (man)power, and interconnections are far more important.

In summary, most methods can be represented by a workflow, which is a collection of connected (dependent) tasks with input, output and required resources. By completing all the tasks of the workflow, one arrives at the final result.

In our time, computers can be used to some extent in virtually all applications, and in this context, a lot of the book-keeping happening between tasks in a workflow can be performed using the computer itself. This allows one to focus on the tasks themselves, while the computer keeps track of results, inputs, and the order of tasks. Colloquially, a program that is able to keep track of the workflow is called a workflow manager.

Quite a selection of workflow managers exists, and in the space of computational materials science, AiiDA, Covalent, FireWorks, Jobflow, and MyQueue are already established workflow engines. They are all capable of running vast workflows with multiple workers and differing computational resources. Despite their usability in many situations, it is still hard to build truly dynamic workflows - where the structure changes while running - especially dynamic workflows that contain any sort of cyclicity of tasks. A particular case is active learning workflows, which have to run the training-selection-labeling until the underlying model is good enough. These workflows are hard to conceptualize in these established "static" managers. Additionally, the procedures for setting up dynamic workflows in these managers are not very intuitive since the workflow structure appears as the workflow is running, which means that making errors in the construction is quite easy.

Together, this results in a relatively high barrier to entry for more complex use cases, where the reliance on purely acyclic dependency graphs requires the user to devise creative ways to implement loops in their workflows.
With this in mind, we have created a different solution in which the user has to specify the structure of the workflow upfront using a small selection of very powerful modular building blocks. This workflow manager is called PerQueue - a contraction of Persistent Queue - and is supposed to be a lightweight self-contained program that runs directly on the resources where it is needed.

2 PerQueue Architecture

PerQueue is written in the Python3 programming language, and it is designed to be a data-passing and persistence framework that utilizes the MyQueue package to submit jobs to the chosen queuing scheduler. The workflows, including the state and data for each task, are stored in an SQLite3 database that is stored locally to where PerQueue runs from. This means that PerQueue is meant to be installed on the (super)computer from where all its workflows will run, though a clever user will be able to start the jobs on remote systems.

Contrary to its competing workflow managers, PerQueue is built to give a static picture of both flexible and dynamic workflows. This means that the workflow structure must be determined upfront, instead of being built only through execution, though it is possible to add more tasks to a workflow after submission. This allows the user to have a clear overview of the workflow already from the point of submission, instead of relying on their own code to correctly and dynamically build the workflow. While this might seem to limit flexibility, this is not the case for PerQueue. The dynamic and flexible nature comes from connecting its workflow structures and running arbitrary code in each step.

In PerQueue, the workflow itself consists of single pieces of work, each denoted as a task and represented by a Task object. For creation, these require the code file to run (enforced to exist), its (known) arguments and the allocated resources. Additionally, a custom name can be given to each Task.

A singular task can be submitted to PerQueue on its own, but the power of PerQueue only really shows when a collection of these are connected within a Workflow. Dependency linking is done by representing the workflow as a directed acyclic graph (DAG), where the roots denote the tasks that are started immediately, while the children only start when their parent tasks have succeeded.

Of the strengths of PerQueue is the level of abstraction that exists with workflow creation. The Workflow object can contain not only Task objects but also other Workflow objects, such that sub-workflows can be specified and reused without incurring memory overheads or code duplication.

Once a Workflow or Task is submitted to the PerQueue manager, the manager runs a static compilation of the workflow, where sub-workflows are unpacked to result in a flat structure as possible. Thereafter, all the unique Tasks are committed to the database together with all DynamicTaskGroups, described below.

The manager then finds all the Tasks that have no dependencies or whose dependencies have been completed. It creates an Entry for each of these and sends the entry to its specified scheduler to run and get the result.

An Entry keeps track of a few things, most importantly which Task to execute, its current state, its PerQueue specific arguments, described later, and once completed, the data that the Task returned. A powerful feature of PerQueue is that the data of Entries is both stored in the database and automatically forwarded to all subsequent tasks. This ensures a flexible and simple flow of data through any workflow. Moreover, it can ensure reproducibility, if the user stores all relevant parameters of the task in the output.

2.1 Statically Resolved Task Groups

PerQueue ships with a handful of modular workflow pieces, which simplify the construction of workflows and are used for setting up the dynamic parts of those workflows.

The workflow pieces are collectively denoted by TaskGroup. These are then divided into two groups, that of StaticTaskGroups which each have functionality at workflow submission, and DynamicTaskGroups which each have dynamic behavior. The latter can only be resolved while the workflow is running, since they represent a choice to be made within the workflow.

At the time of workflow submission, PerQueue runs a static compilation, which entails running recursively through the built workflow. Here, it collects all Task objects and resolves all StaticTaskGroup objects, i.e., it runs their functionality and returns their generated Tasks. While all StaticTaskGroups can be fully resolved at submission, such is their definition - any DynamicTaskGroup instead resolves its contents and then returns itself. In effect, DynamicTaskGroups are stored in the Workflow alongside the Tasks, where they are ready to be processed before being run.

In this way, the static compilation flattens the user-specified workflow down to its simplest possible representation, consisting of only Tasks and DynamicTaskGroups, the latter of which contains only Tasks and possibly only DynamicTaskGroups. This ensures that the workflow that PerQueue has to act on is as simple as possible, while the setup created from the user’s perspective can be as flexible and complex as they want.

2.1.1 Workflow

The Workflow object is used for gathering tasks into a connected collection - the intended workflow. This means that the dependencies between tasks are defined within this object. A restriction here is that the tasks must form a DAG, which simply means that there can be no connections back to earlier tasks - any task must depend on only tasks that are not (eventual) dependencies of itself. This lack of cyclicity is necessary for static compilation, but it can easily be introduced with the use of a CyclicGroup, which is described later.

The secondary job of the Workflow object is to create sub-workflows since these can be given to other TaskGroups, or simply be used for grouping tasks together. Due to the static compilation, this incurs no memory overhead, and since the sub-workflows are flattened by the compilation, it is very useful for workflow setup.

2.1.2 StaticWidthGroup

In cases where the workflow requires doing the same actions to a collection of data, where the only difference between the ac-
tions is their input, one can make use of a StaticWidthGroup (SWG). This takes either a single Task or a Workflow as its sub-workflow. The SWG will automatically create a number of parallel sub-workflows - the number given at creation as the width. Each sub-workflow, denoted as column, is given an index, such that the code can index back into a data array to get the right input.

Through the static compilation, this group is flattened to a simple Workflow, where each Entry contains a piece of data, pq_index, which is the column index of that/ those task(s). Contrary to its dynamic counterpart, the DynamicWidthGroup, this group requires that the width is known at the time of workflow submission, which means it is useful for, e.g., triple determination of an experiment or a set number of endpoints for a Nudged Elastic Band (NEB)\(^9\) calculation.

2.2 Dynamic Task Groups

One of the main features of PerQueue is that of dynamic groups of tasks, which are TaskGroups that are resolved during the execution of the workflow itself. This allows one to include logic in the workflow that depends on the results obtained within the workflow itself.

Currently, PerQueue supports three types of DynamicTaskGroup. These are the CyclicalGroup, the DynamicWidthGroup and the SwitchGroup, each of which is explained below. What they have in common is that they take an input from their dependencies, which determines their behavior in a known way.

2.2.1 CyclicalGroup

The CyclicalGroup (CG) allows a group of tasks to run a number of times until a given condition is met. This functionality is just like a while-loop with a hard iteration limit. When created, the group takes a parameter, max_tries, that specifies the maximum number of times the group of tasks can run before PerQueue stops executing the loop. This hard limit is enforced to ensure that the workflow does stop at some point, raising an error, instead of trying forever on something that will possibly never exit.

Working with a CG requires that at least one of the final tasks of the group returns the PerQueue constant CYCLICALGROUP_KEY as a boolean - true means stop the loop. If multiple tasks return this value, the loop continues if any of them returns false.

2.2.2 DynamicWidthGroup

The DynamicWidthGroup (DWG) is the dynamic version of the SWG, where the number of columns is determined from the task(s) immediately before the DWG in the workflow. This also means that no workflow can begin with a DWG, since there will be no task to give the required data. To give the width to the group, at least one of the dependency tasks must return the PerQueue constant DYNAMICWIDTHGROUP_KEY with the width as an integer, which must be the same if returned from multiple tasks.

Besides the dynamic nature of the DWG, it has exactly the same functionality as the SWG.

2.2.3 SwitchGroup

The SwitchGroup (SG) functions like a block of if-elseif-else clauses - also known as a switch or match statement - where the execution path is determined by the dependencies. At submission, one defines a dictionary of keys and Task or Workflow values, such that each key corresponds to a specific sub-workflow. A restriction here is that an empty path is not allowed due to how the workflows are stored in the database.

To control which path the workflow takes during runtime, at least one of the dependency tasks must return the PerQueue constant SWITCHGROUP_KEY with the key for the given path. To skip the SG - taking the empty path not allowed at creation - one can set the value of the SWITCHGROUP_KEY to the PerQueue constant SWITCHGROUP_SKIP. This acts as if the path was empty skipping the SG. In this way, the no-operation is still possible.

2.3 Filtering Workflows

PerQueue has another capability, which is that it natively implements filtering workflow steps, where each Entry can mark itself as discarded upon finishing. This ensures that none of its dependents get to run since the task itself determined that it did not fulfill its filtering condition.

In this way, workflows in PerQueue can easily reduce resource usage, e.g., high-throughput workflows, where one is interested in performing only the relevant calculations.

3 Use Cases

The following sections show the power and flexibility of PerQueue in real-world cases. These are deliberately chosen to be at different scales to illustrate the wide applicability of the workflow manager.

This starts with a workflow for high-throughput screening with filtering steps using Density Functional Theory (DFT). Secondly, a workflow that utilizes active learning to accelerate the training of machine-learning interatomic potentials. Thirdly, a workflow for cluster expansion and (kinetic) Monte Carlo simulations. Lastly, a workflow for active learning for segmenting electrode images with a human annotator is presented.

3.1 High-Throughput Screening for Solid-State Electrolytes

High-throughput computational screenings serve as robust frameworks for generating novel candidate materials tailored for specific applications.\(^{10,11}\) These screenings begin with the creation of a vast pool of starting systems, which are subsequently subjected to a series of steps that determine whether the material is still a good candidate. This is analogous to a dynamic funnel. Each step operates under defined conditions, allowing qualified systems to progress further along the screening pipeline. As the screening advances, there is an anticipated increase in the computational cost associated with calculations.

In this context, our objective is to explore a fraction of the vast chemical space in pursuit of adept solid-state electrolyte materials by means of DFT. A SWG is used to run the same workflow for three different vacancy locations in the material for each of the starting systems. At each "Conditional" connection in Fig. 1, that branch
of the workflow can decide to stop, using the concept described in Sec. 2.3. The high-throughput aspect is implemented by creating one of these schematics for each of the thousands of decorations available by using different elements on the atomic positions of a prototype material. Consequently, by the final step, the number of candidate systems is significantly reduced, facilitating the utilization of resource-intensive techniques such as NEBs.

To streamline this process, a comprehensive workflow manager proves indispensable. Such a manager should not only facilitate the tracking of past and forthcoming steps but also empower users by enabling automated decision-making based on the generated results. PerQueue stands out in this capacity by abstracting complex workflows into simple parallel graphs, whose data passing allows for enhanced traceability through robust provenance tracking. PerQueue efficiently handles multiple parallel instances, essential for conducting high-throughput computational screenings. Moreover, the inclusion of descriptors within the manager’s framework aids in identifying deviations from desired criteria. This enables users to sift through non-conforming results and reduce unnecessary workload. Armed with this detailed insight, users can make informed decisions despite the large amount of generated data points, thus optimizing the screening process.

In essence, the synergy between high-throughput computational screenings and advanced workflow management tools like PerQueue marks a significant development in material discovery methodologies. By automating repetitive tasks and providing enhanced decision-making capabilities, these tools not only expedite the screening process but also empower researchers to explore novel material landscapes with greater efficiency and precision.

3.2 Active Learning for Machine-Learning Interatomic Potentials using Molecular Dynamics

The use of machine learning in computational chemistry has increased significantly due to its ability to combine the accuracy of \textit{ab initio} methods with the efficiency of classical force fields. The main requirement for these force fields is a high-quality data set. This means additional tasks must be performed before the ML model can be used to calculate properties of the system with traditional computational methods. These extra steps contain data generation and training of the ML model itself. This can lead to complex workflows, e.g., the methodology used in CURATOR for efficient data generation, which can be effectively managed by PerQueue.

CURATOR is an autonomous batch active learning workflow for constructing Machine-Learned Interatomic Potentials (MLIPs). The workflow efficiently explores the potential energy surface (PES) to generate training data by employing four steps. First, a MLIP is trained on an initial data set, which is then used to simulate the PES through methods such as Molecular Dynamics (MD). This is done for a given number of configurations, e.g., structures with different vacancy concentrations. Using batch active learning, informative batches from the exploration are selected to calculate the true DFT energy for these images. The images are added to the training set, such that the model can be retrained. This is continued for several iterations, until the trained model reaches the required accuracy.

To provide a seamless user experience, these four steps, train-simulate-select-label, can be encapsulated in a workflow using the PerQueue engine. \textit{A priori}, it is unknown how many iterations are needed to reach the required accuracy, which the \textit{CyclicalGroup} solves by starting a new iteration if the accuracy is not reached after the model is trained on the new data. Additionally, it might be beneficial to dynamically change which configurations to explore. For example, if the model is extremely accurate for a given vacancy concentration, little new information is gained by exploring that concentration, so it should be stopped. By using a \textit{DynamicWidthGroup} for the simulate-select-label steps the exploration is stopped for well-learnt configurations. Finally, it is common to use an ensemble of models with different hyperparameters for the MLIP to provide a measure of uncertainty. By also using a \textit{DynamicWidthGroup} for the train step, the ensemble size can be updated during the workflow, in case the user wishes so.

Thus, PerQueue both reduces the work required by the user through autonomous handling of task submission, while also giving complete control and flexibility to the user if the workflow has to be modified while in progress.

3.3 Cluster Expansion for Monte Carlo Simulations

Following the ML/MD workflow presented above, one can expand this to even larger system sizes. Considering a system of micrometer scale, we would like to find the most stable configuration and model its dynamics. Doing so is possible through either MD or Monte Carlo (MC) simulations.

The most stable configurations of the system are required for MD simulation to model the dynamics, while MC methods can
both find the most stable configuration and model the dynamics. Using purely ML would be too computationally demanding, due to its nonlinear scaling, which is why this use case employs another method called Cluster Expansion (CE), which shows linear scaling. With a trained CE model, each MC will be performed in microseconds, making CE the ideal tool to perform MC simulations of larger systems.

The workflow illustrated in Fig. 3 is used to efficiently obtain a trained CE model. The workflow autonomously trains a CE model for a particular system, only based on the structure file for that system and the chosen CE model parameters. Importantly one can choose what method should be used to perform geometry optimizations. That can either be pure DFT, which should be used for smaller sizes, or machine learning interatomic potentials. These MLIPs can either be one like the one described in Sec. 3.2 (CURATOR) or a foundation model.

Using the dynamic nature of PerQueue for this workflow allows for on-the-fly limiting the search space of the CE models training. Implementation-wise, this means that the number of randomly generated structures to optimize during each iteration of a CG, as well as the convergence criteria, can be changed dynamically during model training by using a SWG. The dynamic framework also autonomously changes the method used to generate structures for the training, based on the convergence value. The CE model training method can also be changed for each iteration. This makes the training dynamic, which is needed since a purely autonomous CE model training can be truly difficult. Obtaining an accurately trained CE model initiates the MC simulation, where simulations for one or several systems are run in parallel by employing a SWG. This enables the annealing process to be performed efficiently by minimizing the number of MC steps needed for each temperature. A kinetic Monte Carlo (kMC) simulation, using the low energy structure for a specific temperature from the MC simulation, is then run to simulate the dynamics of the structure based on the diffusion barriers of each movement.

PerQueue manages all this within a single workflow, orchestrating the workflow autonomously while allowing for on-the-fly human interaction when needed. Thus, PerQueue is able to autonomously and efficiently train a CE model and perform mesoscale simulations.

### 3.4 Active Learning for Efficient Image Segmentation

With 3D imaging techniques such as X-ray nano-holo-tomography, high-resolution images of electrode material microstructures can be obtained. Being able to quantify these microstructures in terms of the proportions of active and inactive material, as well as in regards to the geometric organization and the development of cracks as the battery is cycled, is important to gain insights into the electrochemical properties of the battery.

To do so, it is necessary to attribute each pixel in a slice to different phases, referred to as segmentation, with high accuracy. Unfortunately, this process is still reliant on expert annotation. This is time-intensive and slows down research. Luckily, deep learning methods for segmentation have reached high accuracy and present a viable alternative. However, they also require large amounts of data. For this task we use a U-Net architecture...
Fig. 3 Schematic for PerQueue workflow for Cluster Expansion (CE) model training and Monte Carlo (MC) simulation workflow. The workflow is initiated with the system's structure file, CE parameters, MC parameters and employed optimization method - either Density Functional Theory (DFT) or a trained machine learning potential - within a single Task. The next part of the workflow is done with a CyclicalGroup, with geometry optimizations within a DynamicWidthGroup - such that the number of optimization in each iteration can be varied. Each geometry optimization is used to re-train the CE model. The CyclicalGroup stops, when the CE model has reached convergence, defined by the cross validation score from the training. Following this a StaticWidthGroup contains MC simulations for low energy structures and by kinetic Monte Carlo (kMC) to simulate the dynamics of interest. The group width corresponds to either several permuted systems or a single system.

Legend

and show that a U-Net trained on an extensive dataset can reach human-level accuracy.\textsuperscript{19}

Active learning is a machine learning algorithm that iteratively adds more data to the training set which maximally benefits the model, greatly cutting down the total amount of data needed.\textsuperscript{20} Naturally, the workflow includes alternating steps of model training, selection of new samples to be added to the training set and annotation of those samples by an expert.

In this process, the deep learning model is initially trained with roughly annotated data. Once the model has converged to a good accuracy on this data, new data points where the model is highly uncertain are chosen to be annotated and added to the training set. The expert annotator then has to annotate the samples, after which training is re-initiated. The training continues until the model has converged and the new accuracy is calculated. The entire workflow is shown in Fig. 4.

By utilizing a CG, PerQueue allows repeating the process until the model is as accurate as required for the subsequent task without needing to set a fixed number of repetitions in advance. The process alternately requires GPU resources and an action by the expert annotator. By being able to switch between processes, GPU resources can be reserved only when they are needed for training and not in the time intensive step of labeling. If multiple networks are to be trained for an ensemble of neural networks, this can easily be done without needing to restructure the process by simply introducing a SWG and a DWG in the workflow.

Since the annotation is reliant on a human expert, the task called Label would currently involve a waiting loop that checks whether the annotations have been added to the respective folder. If the task were to involve both a human and computational resources, e.g., to utilize a pre-trained model for segmentation with anchor points to accelerate the high-accuracy segmentation, it would be advantageous to have the human task incorporated explicitly in the workflow.\textsuperscript{21} Similarly, it would be necessary if the human task has an expected waiting time exceeding the maximally allowed process time.

4 Outlook

Beyond the examples shown above, which comprise materials modeling and artificial intelligence, the design of PerQueue is able to leverage the power of Materials Acceleration Platforms (MAPs). MAPs disruptively accelerate the materials discovery by AI-orchestrating theoretical and experimental capabilities from many different sources and sites.\textsuperscript{22,23} In MAPs, the next experiment or simulation depends on the result of the previous step. This means that, to be efficient, MAPs require a dynamic workflow able to interact with different techniques and to handle data from various sources.\textsuperscript{24} PerQueue is one of the first examples of this. In conjunction with systems like FINALES,\textsuperscript{25} PerQueue could be used to increase the autonomy of the whole system by having complex workflows run dynamically with minimally required user interaction.

As can be seen from the above sections, PerQueue is already a quite capable piece of software for managing and running flexible
and dynamic workflows, but it could be improved further.

One of the things that could still be implemented into the program is the concept of a ManualTask, in which the workflow pauses and waits for user input. This could be very powerful in connection with workflows that contain both computational resources and laboratory or other manual labor, where a human is required to be in the loop. Currently, this can be handled by a waiting loop within the task that only exits when the user gives it some input - the result of the manual task.

Additionally, PerQueue is a young piece of software, and where its competitors can run asynchronously with many service providers to get results from many different places, PerQueue is stuck on its local system, which makes remote calls impractical. Allowing PerQueue to live as a service, that can communicate with many providers could be very interesting. At the moment, PerQueue is also very much tailored for single user workflows, so, it could use better capabilities for multi-user workflows.

5 Conclusions

Here, we have introduced the software PerQueue, which is a new workflow manager written in Python3. PerQueue is a highly flexible workflow manager that focuses deeply on dynamic workflows, whose structure is defined up front to reduce the number of mistakes generally encountered in dynamic workflows. The concept of PerQueue, and the method by which it achieves its goals, is by treating the workflow setup as a building blocks session, where predefined modular blocks connect to make a full workflow.

These building blocks allow PerQueue workflows to exhibit cyclicity, dynamic width of parallel sub-workflow and powerful conditional control, with filtering workflows natively supported.

All this is backed by a local database that stores the state and returned data of all the tasks of the workflow and keeps track of the dependency graphs that prescribe the order of operations.

Having introduced this new piece of software or workflow management, its usage in four different computational use cases has been showcased. These deliberately cover multiple scales encountered within materials discovery. They go from a high-throughput study with DFT over MLIs trained with active learning for MD and kMC applications within extended systems, and ending with image segmentation of microstructures of synthesized materials accelerated by active learning ML.

In summary, the authors hope that PerQueue will aid multiple researchers in accelerating their work processes in a simple but powerful way, leveraging easily accessible dynamic functionality and data passing between each task.

Author Contributions

Benjamin H. Sjølin: Conceptualization, Methodology, Software, Writing - original draft, Visualization. William S. Hansen: Validation, Investigation, Writing - original draft, Visualization. Armando A. Morin-Martinez: Validation, Investigation, Writing - original draft, Visualization. Martin H. Petersen: Validation, Investigation, Writing - original draft, Visualization. Laura H. Rieger: Investigation, Writing - original draft, Visualization. Tejs Vegge: Resources, Supervision, Funding acquisition, Writing – review & editing. Juan M. G. Lastra: Supervision, Funding acquisition. Ivano E. Castelli: Conceptualization, Visualization, Supervision, Project administration, Funding acquisition, Writing – review & editing.

Conflicts of interest

There are no conflicts to declare.

Acknowledgements


WSH and JMGL acknowledges that this project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Grant Agreement No. 101054572).

LHR acknowledges support from the Energy Technology Development and Demonstration Programme (“ViPES2X : Fully AI-driven Virtual Power Plant for Energy Storage and Power to X”).

TV acknowledges funding from the Pioneer Center for Accelerating Materials Discovery (CAPeX), DNRF Grant P3.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 957189 (BIG-MAP). It is also part of BATTERY 2030+ initiative under grant agreement No. 957213.
References


18. P. Canepa, ACS Materials Au, 2022, 3, 75–82.


