SAI: AI-Enabled Speech Assistant Interface for Science Gateways in HPC

Pouya Kousha(✉), Arpan Jain, Ayyappa Kolli, Matthew Lieber, Mingzhe Han, Nicholas Contini, Hari Subramoni, and Dhableswar K. Panda

The Ohio State University, Columbus, OH 43210, USA
{kousha.2,jain.575,kolli.38,lieber.31,han.1453,contini.26}@osu.edu,
{subramon,panda}@cse.ohio-state.edu

Abstract. High-Performance Computing (HPC) is increasingly being used in traditional scientific domains as well as emerging areas like Deep Learning (DL). This has led to a diverse set of professionals who interact with state-of-the-art HPC systems. The deployment of Science Gateways for HPC systems like Open On-Demand has a significant positive impact on these users in migrating their workflows to HPC systems. Although computing capabilities are ubiquitously available (as on-premises or in the cloud HPC infrastructure), significant effort and expertise are required to use them effectively. This is particularly challenging for domain scientists and other users whose primary expertise lies outside of computer science. In this paper, we seek to minimize the steep learning curve and associated complexities of using state-of-the-art high-performance systems by creating SAI: an AI-Enabled Speech Assistant Interface for Science Gateways in High Performance Computing. We use state-of-the-art AI models for speech and text and fine-tune them for the HPC arena by retraining them on a new HPC dataset we create. We use ontologies and knowledge graphs to capture the complex relationships between various components of the HPC ecosystem. We finally show how one can integrate and deploy SAI in Open OnDemand and evaluate its functionality and performance on real HPC systems. To the best of our knowledge, this is the first effort aimed at designing and developing an AI-powered speech-assisted interface for science gateways in HPC.

Keywords: HPC · Open OnDemand · Conversational AI · Speech recognition · Natural Language Processing · Knowledge Graphs

1 Introduction and Motivation

High-Performance Computing (HPC) is an integral part of various traditional scientific domains like medical research, weather forecasting, and earthquake prediction, as well as emerging areas powered by Deep Learning (DL) and Machine

This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

A. Bhatele et al. (Eds.): ISC High Performance 2023, LNCS 13948, pp. 402–424, 2023.
https://doi.org/10.1007/978-3-031-32041-5_21

Learning (ML). The ability to process and analyze large sets of data on current HPC systems has led to remarkable advances in science and engineering and has become an indispensable tool for students, researchers, and industry professionals. Examples include social scientists reviewing massive datasets from sources such as Twitter or Facebook, archaeologists experimenting with LiDAR [18] in mapping subsurface artifacts, and painters harnessing computer-aided design to use archives of ancient works as a style guide.

Unfortunately, HPC use and adoption by many is hindered by the complex way in which these resources need to be used. Utilizing HPC services requires familiarity with command-line interfaces and custom client software of HPC middleware, DL/ML frameworks, and performance analysis tools which creates an accessibility gap that impedes further adoption. For instance, HPC middleware like high-performance MPI libraries and DL frameworks have various advanced features and complex user interfaces. While these interfaces are comprehensive and extensive, they require a *steep* learning curve, even for expert users, making them nearly impossible to use for novice users like medical doctors, domain scientists, and other users whose primary expertise lies outside of computer science.

Recent surveys conducted by supercomputing centers [24] indicate that users are more likely to adopt a GUI-based interface provided by science gateways such as Open OnDemand [5]. Open OnDemand is one of only a few open-source general web interfaces to support remote visualization. It is currently the most well-known and adopted general web interface within the HPC community. Although, Open OnDemand reduces the initial accessibility hurdle to the HPC ecosystem by providing job templates for a small subset of popular HPC applications, there is still much to be desired in extending this support to the other components of the HPC ecosystem such as middleware, frameworks, and tools. While most users are intuitively able to express what they are looking for in words or text (e.g., "train my model with 32 GPUs on TACC Frontera with TensorFlow"), they find it hard to quickly adapt to, navigate, and use HPC interfaces to obtain desired results. Furthermore, surveys of end users conducted by prestigious firms like Deloitte [4] and PriceWaterhouseCooper [6] clearly indicate that users are more likely to use a conversational AI interface as opposed to using older keyboard/mouse-style inputs. **To the best of our knowledge, no interface exists that allows end-users to interact conversationally with state-of-the-art science gateways**.

1.1 Motivation

This challenge leads us to the primary motivation of the proposed work: ***can we design an easy-to-use and productive conversational interface, utilizing AI, that enables end-to-end abstraction and automation of the steps involved in execution, monitoring, and evaluation of HPC workloads?*** Fig. 1 depicts our vision of how SAI enhances the productivity of end-users. The left side depicts the multiple steps that the end users must traditionally perform to execute their HPC applications. These steps typically include selecting an HPC application; figuring out dependencies and installing them and the actual HPC application (either manually or through package managers); consulting the documentation for appropriate arguments and parameters; and finally creating the job launch scripts. All of these steps are complex and require expertise in interacting with HPC middleware and tools using their traditional interfaces. The right side depicts how these same users can extract better productivity by using SAI. The manager of HPC at the U.S Department of Energy Idaho National Laboratory had the following strong and enthusiastic statement for SAI work - *"We have seen early demonstrations of the conversational AI Engine on multiple occasions. We see the proposed work as a paradigm shift that will directly benefit the over 1,200 users on our systems and lower the threshold for HPC usage. The incorporation of the AI Engine in a science gateway will serve to lower the time to science for the vast majority of our HPC users."*

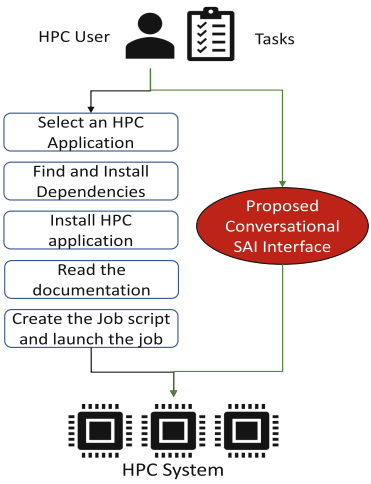

Fig. 1. Motivation behind creating SAI to improve user productivity

1.2 Challenges in Enabling Conversational Interface for HPC

Challenge #1: Creating Custom Datasets and Models for HPC: While the latest Automatic Speech Recognition (ASR) [9,27] and Natural Language Understanding (NLU) [10,23] models have achieved impressive accuracy rates, such as 2% Word Error Rate (WER) [9] on Librispeech [21], these models often struggle to accurately interpret and understand technical terms (e.g. Allreduce and MNIST) and abbreviations (e.g. CPU and HCA) specific to the HPC domain. Furthermore, current language datasets do not include these technical terms and abbreviations, making it difficult to create ASR and NLU models that can accurately interpret and understand words and sentences commonly used in HPC. Thus, the availability of datasets specifically tailored to HPC domain is crucial and is key to creating new NLU/ASR models capable of accurately interpreting HPC-specific words and sentences. *To the best of our knowledge, HPC-specific datasets and models do not exist for use today.*

Challenge #2: Scalable Representations for Complex Relationships between Components of the HPC Ecosystem: The relationship between HPC applications, parallel hardware, deep learning (DL) models and problems, and datasets/inputs is complex with respect to each other. Researchers currently spend significant time and energy manually understanding and mapping these relationships through the use of documentation, tutorials, and other online resources. However, the representation of these complex relationships can be automated and made more accessible to end-users by leveraging Knowledge Graphs (KGs). An essential aspect of creating such KGs is the use of a portable, simple, yet thorough ontology. Recently, the HPC Ontology [17] has been proposed as a way to formally define and represent HPC-related knowledge, including vocabularies, semantics, and formal representations. *However, it only captures limited aspects of the complex relationships we want to cover for HPC workload execution. Thus, a significant expansion and enhancement is required for the proposed workflow to be truly useful.*

Challenge #3: Automating and Abstracting Installation of Packages: Leveraging High-Performance Computing (HPC) systems requires the use of various libraries, middleware, and applications. MPI implementations such as MPICH, OpenMPI, and MVAPICH2 enable parallel computation at scale. System software like compilers and supporting libraries are vital for accelerating applications. Frameworks like PyTorch and TensorFlow provide high-level APIs for designing and training deep neural networks. However, installing these software packages and their dependencies is a significant challenge, even for those familiar with HPC systems. While package managers such as Spack simplify the process, they can still pose a challenge for novice HPC users whose primary expertise lies outside of computer science.

Challenge #4: Integration of Conversational AI to HPC as Gateway: Developing a conversational interface framework is only the first step, the next challenge is to integrate it into a state-of-the-art science gateway to provide end-users access to it. To achieve this, we need to determine the interface between the conversational AI interface and the science gateways. The conversational interface component must also be modular to adapt to future advancements in DL models and HPC applications without a major revamp. A challenge here is to ascertain and minimize the changes needed to enable the end-to-end pipeline.

1.3 Contributions

In this paper, we take on the challenge of reducing the complexity of executing traditional scientific and ML/DL-based HPC workloads through modern science gateways by proposing, designing, and developing SAI. SAI is a novel conversational AI-based framework that automates and abstracts the cumbersome steps involved in accelerating traditional scientific and ML/DL-based applications on modern HPC systems. SAI simplifies the HPC process for non-experts, such as

domain scientists and AI researchers. It eliminates the need to learn about different job queues on a cluster, allowing them to focus on their research without bogged down by technical issues, such as having to learn about the various job queues on a cluster. With SAI, these researchers can easily submit their jobs and get the results they need, without needing to become experts in the intricacies of HPC systems.

To gain a deeper understanding, users can familiarize themselves with the transition flow of SAI (Fig. 7). This will enable them to reproduce results on the terminal using the SAI-generated command as discussed in Sect. 5. To summarize, this paper makes the following contributions:

1. Proposes and develops a conversational AI interface (SAI) for running HPC applications and installing required libraries, packages, and frameworks
2. Describes datasets for text and speech with HPC-specific and fine-tuned state-of-the-art ASR models to recognize HPC terminologies and retrain an Entity-detection NLU model to understand text command
3. Proposes a general ontology to scalably represent the complex relationships between various components of the HPC ecosystem
4. Describes KGs to represent the relationship between different scientific/DL benchmarks/applications, datasets/inputs, package managers, and tools.
5. Integrate and deploy SAI in Open OnDemand and evaluate its functionality and performance on real HPC systems.
6. Provides a comprehensive explainable flow for SAI, including a detailed explanation of the transition from user input to job output along with generated job scripts and installed environments. This helps users to understand how to generate commands and reuse them directly in a terminal in future.

2 Background

2.1 Conversational User Interface

Conversational User Interfaces (CUI) represent a new way for users to interact with applications, moving beyond the traditional Graphical User Interfaces (GUI). Popularized by voice assistants like Siri, Alexa, and Google Assistant, CUIs have the ability to understand and respond to multiple variations of natural language, enabling more intuitive and efficient communication. Studies [12,13] have shown a strong preference for speech interfaces over traditional GUIs due to the ease of use and minimal learning curve. The use of CUIs is becoming increasingly popular in businesses [25] and Data shows that more than half of US adult mobile phone users use virtual assistants such as Siri or Alexa. [3].

2.2 Open OnDemand

Open OnDemand is an open-source, widely-used, customizable web interface for interacting with HPC systems. It allows integration with various HPC resources and job schedulers to make HPC resources more accessible to users who may not

be familiar with command-line interfaces. It has features such as job submission, file management, and remote visualization, providing a streamlined and user-friendly experience for researchers, engineers, and scientists.

2.3 Ontology and Knowledge Graphs

Ontology formalizes knowledge of entities in a domain with limited relationships and classes for constructing KGs by adding individuals and instantiating the data and object properties. The data property applies to an individual to capture features or data about the individual while the object property (relationship) links individuals of the same or different classes to each other.

2.4 Spack

Spack is a package manager primarily designed for HPC systems, providing flexibility in build configuration and high compatibility with different systems. It builds packages and dependencies from source, allowing customization without interacting with build systems or resolving dependencies. Users interact with Spack through "specs" specifying package version, compiler, features, and dependencies, which Spack verifies before proceeding with installation.

3 Terminologies

The various terminologies, terms and legends used in this paper are explained below. A parameter is a value that is given by the user for an argument and arguments can have multiple parameters.

- **Entity**: a single or a collection of words that refers to a same class always. For examples, allreduce and ResNet are algorithms/model.
- **HPC-ASR Dataset**: an in-house ASR dataset created by us for HPC and DL terminologies.
- **HPC-NLU Dataset**: an entity detection and classification dataset created by us for training NLU models for HPC and DL terminologies.
- **Speech and Text Query**: Speech query is spoken audio passed to ASR and NLU models, text query is typed text passed to NLU model.
- **WER**: WER is a performance metric for ASR models that works by comparing words in the predicted and the reference text.

4 Proposed SAI Framework

In this section, we elaborate our design and implementation to enable the Speech Assistant Interface (SAI) for the HPC domain. Figure 2 depicts the overall flow of execution and the steps involved in the operation of SAI. We will describe each step in the following subsections.

4.1 Generating HPC Datasets for Speech and Text

To address Challenge-#1 (Sect. 1.2), we create an HPC-datasets for text (**HPC-NLU**) and speech (**HPC-ASR**) containing HPC and DL terminologies (for example NCCL, IntelMPI, ResNet, etc.). ❶ We generate basic text queries and label each entity into five broad categories of model/algorithm, data, system, software, and arguments.

We create a list of arguments that can be given to the application to generate different types of queries on the HPC text dataset. ❷ We generate all the combinations of entities with different arguments for each basic sentence structure. For example, the number of combinations of the commands for running MPI benchmarks amounts to 315K queries. ❸ To handle different ways of saying the same phrase, we develop synonyms for HPC terminologies (like CPU, processor, central-processor, and host-processor for CPU) and use them to generate additional queries. The mentioned MPI benchmarks query set extends to 19 million queries by using the synonyms. These queries will cover most of the HPC lexicon and for HPC-ASR we crowd-sourced to 20 different volunteer users—with 6 dialects and speech patterns— recording portions of it to create the HPC-ASR dataset. ❹ We include permutations of phrases to restrict DNN from learning any ordering of arguments in the dataset. The resulting MPI benchmark dataset contains 7 million rows just by including the permutations. Through each step, the labels and queries of both HPC-ASR and HPC-NLU are human supervised. For HPC-ASR, the accents are covered by the TIMIT dataset for training. The recordings are denoised and verified through human supervision. Using the five broad categories mentioned, the entities are classified to these 5 types and are passed to the NLU component for processing and value extraction. Section 6 mentions train-test data split details.

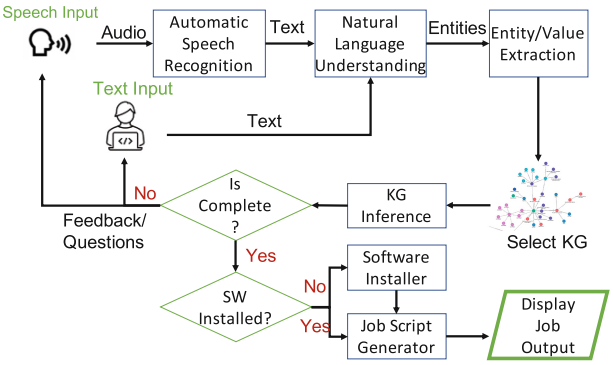

Fig. 2. High-level design of SAI showing the flowchart and SAI components - The blue rectangles are components of SAI while the green boxes show the decision criteria for the direction to proceed based on the processing of user input to continue interacting with the user or moving toward submitting the associated user job. (Color figure online)

4.2 Fine-Tuning Speech Recognition Model for HPC Terminologies

As the first step of processing speech input shown in Fig. 2 and to address challenge-#1 (Sect. 1.2), we need an ASR model capable of understanding

domain specific terminologies (e.g. PyTorch, Allreduce, and IntelMPI) in HPC/DL applications. State-of-the-art ASR models are trained on large speech datasets like LibriSpeech and TIMIT to recognize English's large vocabulary and support different accents. We selected Speech2Text [27] as the base model, pre-trained on the LibriSpeech 1,000 h ASR corpus. To achieve our goal, we combined the HPC-ASR dataset with TIMIT [1] and fine-tuned the model and hyperparameters. TIMIT dataset helps supporting different English dialects and accents. We convert the generated text from the ASR model to lowercase and used SentencePiece [16] to tokenize the words to be passed to the NLU module.

4.3 Designing an Entity Detection and Classification Model for SAI

The next step depicted in Fig. 2 is to apply natural language understanding on user text input or transcribed text from ASR to overcome the rest of challenge-#1 (Sect. 1.2). Therefore, we designed a BERT-based entity detection and classification model [11] to extract entities and classify them into five broad categories: model/algorithm, data, system name, software, and arguments to understand and execute the given command. Figure 3 shows the architecture of the proposed DL model used to detect and classify entities in a sentence. To support multi-word terms, we label the first word as B-Category-Name and consecutive words as I-Category-Name. Since arguments can have a numerical value, we create key-value pairs for the argument category by post-processing the NLU output. Arguments could have floating point values; therefore, we support numbers in numerical format only (for example we support "4.56" not "four point five six"). The output of this module is a dictionary of entities with their assigned values like ('Model':'Inception3'). This list is used to query KGs in the next step.

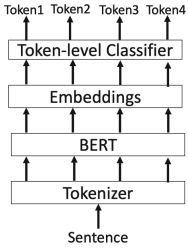

Fig. 3. Proposed BERT-based entity recognition model for SAI

4.4 Creating the HPC Ontology and Knowledge Graphs

The existing ontologies in Sect. 8 do not capture the relationships between HPC components for executing workloads. Hence, to address challenge #2 in Sect. 1.2, we need to create an ontology capable of capturing complex dependencies and the workload relationships between HPC components like systems, software, models/algorithms, data, and their related arguments to construct a complete and useful Knowledge Graph(KG) for different HPC applications. We create a new ontology–called **SAI-O**– with 5 major classes of system, software, model, data, and argument to represent HPC components. Argument has 3 subclasses of software_arg, model_arg, and system_arg. Software has 3 subclass of framework, compiler, and library. A subset of relationships are listed in Table 1.

Table 1. Major object properties in SAI-O ontology

Relation Property	Domain	Range	Description
canBe	any	any	Defines possible values (OR)
runs	any	Software or Model	Captures run capability
depends	Software	System	Captures software dependency
needs	any	any	Defines requirements (no default)
hasArgs	any	Argument	Defines optional values (defaults)
hasSoftware	any	Software	Captures software availability

SAI-O contains data properties like "*version, hasDefault, default, name, description*" that are common between all the individuals in SAI-O. For example, the data property "description" gives a description of the individual to provide further information upon user requests. There are some data properties specific to a class of objects. For example, for a queue class that represents system job queue, we have"size, timeLimit, maximumJobSize, and maxUsableMemory" data properties to describe a job partition information. Note that not all the properties need to have a value. *Due to the lack of space, only a subset of relationships and data properties are shown in the paper.* Through defining standard and generic"classes/relationships/data" properties in SAI-O, we can capture different asserted and inferred relationships among HPC system, software, model/algorithm, data, and their arguments and query later. SAI-O ontology could be used to add additional HPC applications in the future (Sect. 7.4.)

Using SAI-O, we created the KGs for 3 different HPC applications as a proof of concept in RDF/XML format: OMB Benchmarks [19], Distributed DL training, and NAS parallel benchmarks. In our KGs, synonyms are connected using the "Same individual as" relation to each other. An example of the constructed KG for the Inception3 DL model is shown in Fig. 4 to show the requirements and dependencies to run Inception3 model where green arrows show possible arguments and grey shows "needs".

4.5 Knowledge Graph Selection and Inference

SAI has one KG per application and can support multiple applications. To select the appropriate query for the given query, We define SPARQL [2] queries to query all the available KGs and see which one gives the *max hits* – which KG has the maximum number of entities detected in the given query. We query and process the selected KG to assemble a list of required arguments with their possible values and optional arguments with their default values (Defaults are stored in the KG). The assembled list is compared to the processed user input list if the required parameters are not complete, SAI generates corresponding questions/feedback and interacts with the user back and forth to get the parameters. If a necessary argument has a list of parameters (for example dataset values for Inception3), SAI displays the list to the user to select from it. Otherwise, SAI

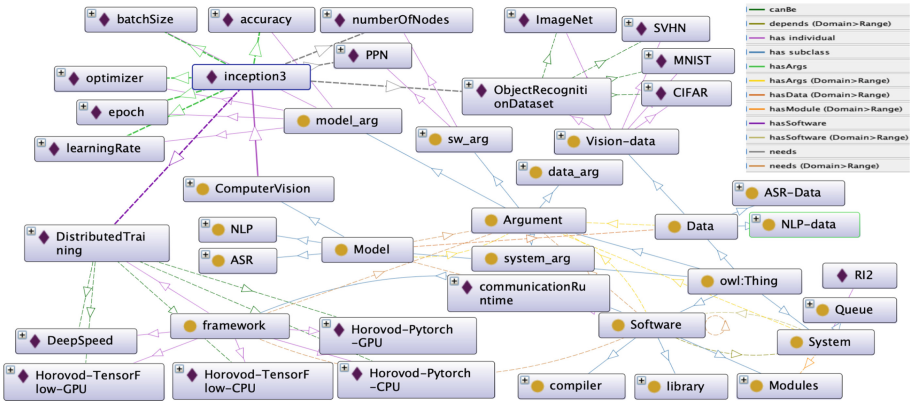

Fig. 4. Screenshot of visualizing Inception3 DL model relations in SAI in constructed DL knowledge graph based on SAI-O ontology - This is only one of the models supported in SAI DL KG. The type of relations are shown at the top left. The yellow rectangles represent classes and purple ones represents individuals in KG (Color figure online)

asks the user to enter the value for a required argument. At the end, we query the KG to get the software dependencies and libraries, which is used in Software Installed module (Sect. 4.6).

4.6 Software Installer Check and Interfacing with Spack

After evaluating the completeness of the user's input, SAI needs to check if the necessary software and packages are in place through the Software Installer component (shown in Fig. 2) to execute the query. For this objective (challenge # 3 described in Sect. 1.2), SAI takes advantage of Spack to resolve installation dependencies, install the requirements, and provide the path of the executables to the Job Script Generator. To enable efficient interaction with Spack, we developed the *Spack Interfacing Layer (**SIL**)* using Spack's python APIs. To avoid the conflicts with system/user-level Spack environments, SIL utilizes a user-specified directory for software installations and its own configuration file that contains all Spack environments, files, and software installations.

To maintain proper dependencies and correctly bundle software and packages for installing, SIL creates a single Spack file by gathering dependency information about each package and combining them into one spec. The installed Spack environments through SAI can later be activated using Spack when the user wishes to do testing outside of SAI. SAI also reuses these environments if they are compatible with new user requests, in order to avoid redundant environments. SAI uses separate, logical environments that can share installations, ensuring that software is only installed if it does not already exist within SAI.

To increase efficiency and prevent system blockage during installations, SIL implements a multi-threaded installation queue and asynchronous installation.

This allows users to request multiple jobs without SAI being blocked while waiting for installations to complete, even for complex packages like Horovod which may take an hour to install.

4.7 Integration with Open OnDemand

To address challenge #4 in Sect. 1.2, In this section we describe the integration of SAI with Open OnDemand Open OnDemand supports two modes of deployment for applications — "Passenger Apps" and "Interactive Apps" as shown in Fig. 5. Passenger applications run on login nodes and resources are shared among multiple users with a separate directory for each user. Interactive applications, on the other hand, run on top of a node allocation to ensure exclusive resources. For the integration, the authors create YAML files to capture system-level information such as job scheduler, number of available nodes, and partition/queue list. For both deployments, we developed SAI setup scripts and job scripts. For Interactive deployment, we modified the Open OnDemand interface for node allocation to include SAI as an application, configured the cluster to enable running interactive SAI on compute nodes, passed user configuration to the job script, and developed scripts for pre- and post-processing.

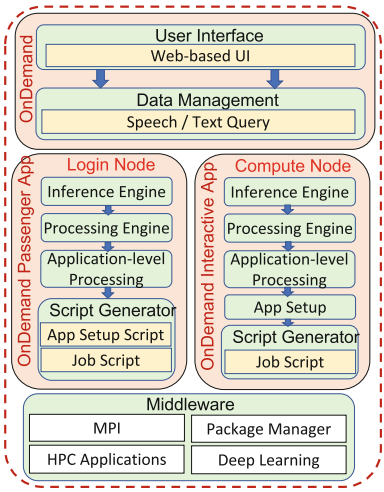

Fig. 5. Integrating SAI with OnDemand

The Passenger SAI application generates scripts for installing dependencies and executing tasks on the login node, while the Interactive SAI application handles dependencies installation on the compute node and submits the task for execution. We also utilize Open OnDemand's job template method to enable the creation of user-defined templates generated by SAI's job script generator. In the future, we plan to generate RPMs, Singularity images, and Kubernetes containers for distributing SAI through Open OnDemand.

5 Insights into SAI Usage and Explainable Flow

Following the flow in Fig. 2, we describe SAI's usage after the integration to OnDemand (Sect. 4.7) to run applications and install dependencies on an HPC system. We describe how SAI addresses the challenges in Sect. 1.2.

❶ Users can access SAI through OnDemand gateway that handles user authentication and remote CLI/GUI connection to SAI. The user selects passenger or interactive deployment of SAI to run. (Challenge #4 in Sect. 1.2)

❷ The user can give tasks using the chat box shown in Fig. 6. SAI converses with the user in natural language to understand their requirements. SAI's chat interface provides a "Mic" button to record the speech command. *SAI does not force a user to use the speech interface every time, it also has a text-based chat interface for users concerned about sending voice.* SAI converts speech to text using ASR model and interpreting the ASR output text or user's text commands with NLU model to prepare commands for HPC application execution, including compilation, running and monitoring. (Challenge #1 in Sect. 1.2)

❸ SAI uses HPC specific ontologies and Knowledge Graphs (KGs) derived from them to assess the user's commands for completeness and correctness. Through the use of these KGs, if SAI realizes that the information provided by the user is not complete, SAI can either use default values for missing information or interacts with the end-user again to get the needed information. (Challenge #2 in Sect. 1.2)

❹ Once the user's input has been obtained and validated, SAI executes the end-user's HPC application on the available hardware resources. Under the hood, SAI installs the application and necessary dependencies, executes the workload, monitors the progress, and reports the results of the application's execution. Users can give the path to the pre-installed software too. (Challenge #3 in Sect. 1.2)

Figure 6 show an example interaction of a user with implementation of SAI where the user tells SAI all the essential parameters in the initial input and changes a parameter. SAI selects and incorporates OMB's Knowledge Graph–based on max hits among all KGs – to validate input accuracy, checking for errors, inconsistencies, or missing information resulting in reducing the risk of errors in the job execution. With all required parameters provided by the user, SAI engages with the user for confirmation and prompts for potential modifications before submitting the job.

SAI Features: SAI's main features include job script generation, job execution, the ability to run jobs on multiple nodes with different architectures including GPUs, and OnDemand integration. SAI automatically finds package dependencies based on HPC system and architecture and supports package instal-

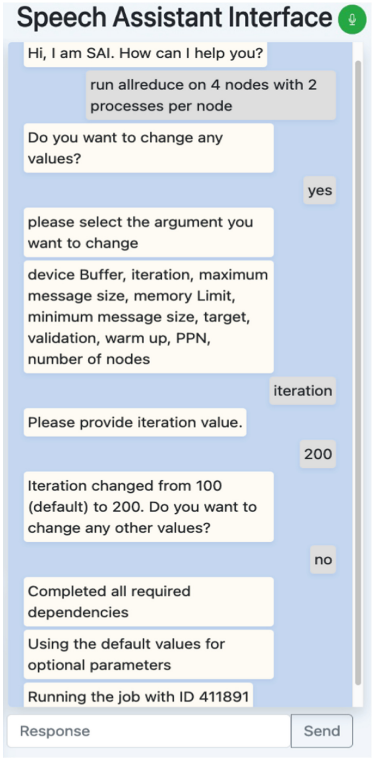

Fig. 6. Screenshot of an user interaction with SAI- the user is running Allreduce benchmark from OMB and changing default values before submitting the job.

lation as well as verification. For applications and job variables, SAI provides the default values, their descriptions, completeness check, and argument validity to reduce the likelihood of errors. The frequency of using SAI chat/speech interface depend on the user's HPC needs. It can be used whenever they want to build or run applications on an HPC system, as well as for tasks such as scaling and job submission. The natural language interface is both user-friendly and accessible, which may encourage more frequent interactions with HPC systems, ultimately resulting in increased overall HPC usage.

Insights into SAI's Flow: Transparency in the internal workflow and output of each component in SAI, from input to output, is of utmost importance. This transparency fosters user understanding and trust, ensuring that SAI is executing tasks as intended. Moreover, it can familiarize new HPC users with the process by showing them the steps taken to understand the flow and reproduce results. To achieve this, we have developed an interface within SAI that offers insights into the transition and output of each component, accessible with a simple click. Additionally, the interface displays the total and component-specific latency for SAI, providing further insight into its performance and enabling users to evaluate its efficiency. Figure 7 illustrates the complete transformation of the user text query initially mentioned in Fig. 6 to job output. In Fig. 7-①, the generated entities from NLU are presented, which are then processed and forwarded to the knowledge graph selection module. Figure 7-② displays the output obtained after querying the KG, revealing a dictionary with required arguments labeled as "need", optional values identified as "defaults", and the selected KG presented to the user. Figure 7-③ showcases the listed –parameters following the processing of the KG query–including required and optional values, along with details of the Spack environment path and working directory path. These parameters are transferred to the job composer by SAI for the creation of the job script. Then, as depicted in Fig. 7-④, the software installer ensures the installation of necessary packages, exhibits the executed commands, and provides users with an option to verify the successful installation of all binaries.

Figure 7-⑤, demonstrates the generated job script by SAI to execute the users job. SAI simplifies the process of submitting jobs to an HPC cluster by creating a batch script and a Spack environment. Once familiar with the process, users can submit jobs directly using the job script and Spack environment generated by SAI. While SAI streamlines the process using SAI is optional, and users are free to use the command line instead with SAI's generated commands and scripts. Figure 7-⑥ displays the final job output of the user's request.

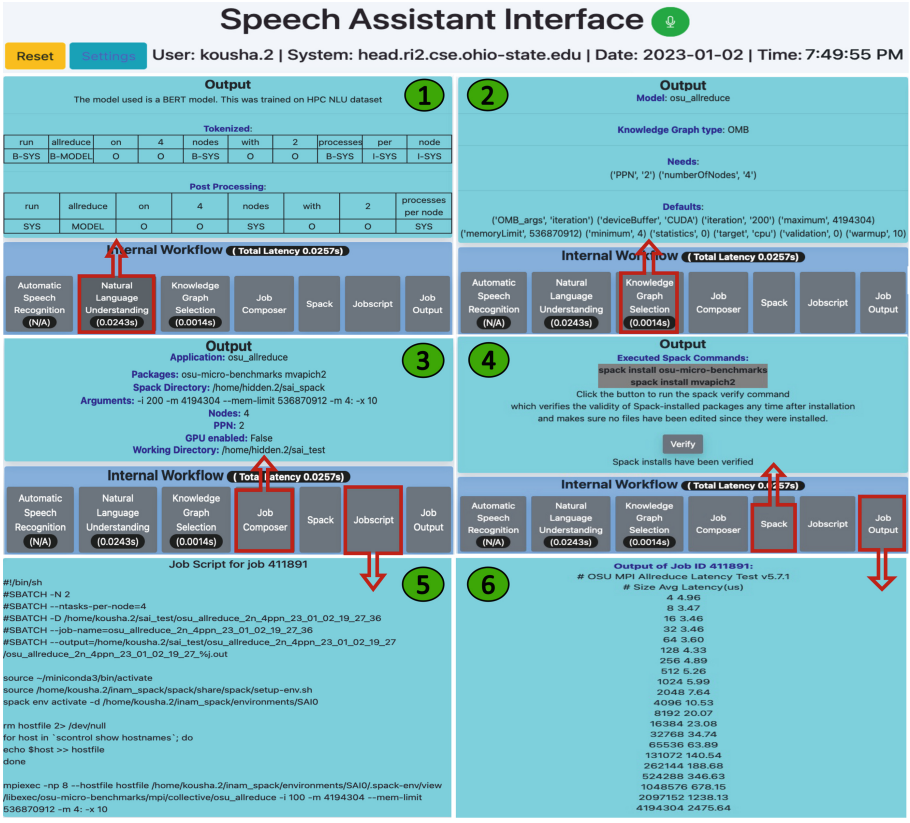

Fig. 7. Visual representation of SAI's implemented pipeline for the user input from Fig. 6: showing series of transformations of through various SAI components, with each step generating an output. The screenshot provides a clear illustration of the flow of data and the transition of input to output at each stage.

6 Experimental Evaluation

6.1 Evaluation Platform

We conduct our evaluation experiments on a 58-node Infiniband EDR cluster. It has two sets of nodes: 1) Intel 28 cores Broadwell(BDW) CPU running at 2.40 GHz nodes with a single NVIDIA Volta V100-32 GB GPU, and 2) Intel 28 cores SkyLake(SKX) CPU running at 2.6 GHz node with two NVIDIA K80 GPUs.

DL Framework: PyTorch [22] defines and trains DNNs for ASR and NLU
DNNs: Speech2Text [27], BERT-based entity detection and classification [11]
Datasets: LibriSpeech [21], TIMIT [1], HPC-ASR, and HPC-NLU

6.2 Evaluation Methodology

In this section, we describe our evaluation methodology used to conduct experiments. In Sect. 6.3, we compare the performance of the existing pre-trained Speech2Text model and fine-tuned Speech2Text (Sect. 4.2) on the HPC test dataset. Then, we test the NLU model trained from scratch – since there is no pre-trained NLU available for HPC – in Sect. 6.4 to predict the entities for the given text query or speech query transcript. The end-to-end performance of the ASR and NLU model is evaluated in Sect. 6.5. Section 6.6 provides the overhead of running the SAI pipeline from deep learning inferencing to determining whether the requested software is installed or not. We evaluate the scaling of SAI as an Open OnDemand Passenger App in Sect. 6.7 and the performance of SAI as an Open OnDemand Interactive App in Sect. 6.8.

6.3 Evaluating ASR Model

We evaluate the performance of pre-trained ASR Speech2Text model on our HPC-ASR dataset. Our HPC-ASR dataset has HPC terminologies and TIMIT dataset has different accents, which will make our proposed design available to a wide range of speakers. The final test results for WER on TIMIT and HPC test set is shown in Table 2. We observed that the existing off-the-shelf ASR model is not suitable for SAI conversational needs as it does not recognize HPC-related terminologies in the test set resulting in high WER. This motivated us to fine-tune our ASR model. Using our fine-tuned ASR model, we were able to improve the performance on the HPC-ASR test set and achieved a better WER, closer to that of state-of-the-art models. Figure 8 shows the fine-tuning of Speech2Text on HPC-ASR + TIMIT datasets. Eval WER is the WER for the ASR model on the validation dataset and Eval Latency is the runtime of one step in ASR model validation.

Table 2. Evaluation of ASR model using Word Error Rate (WER) - Lower is better

Train Dataset	Test Dataset	WER
Base (LibriSpeech)	HPC-ASR	86.2
Base+TIMIT+HPC-ASR	HPC-ASR	**3.7**

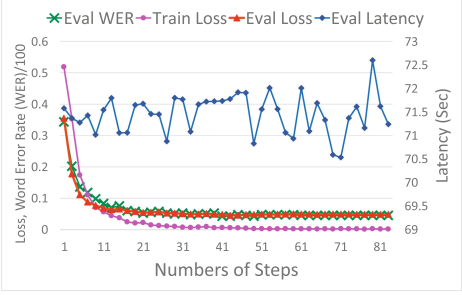

Fig. 8. SAI's ASR Model evaluation for Loss, WER and Latency

6.4 Evaluating NLU Model

Since no pre-trained NLU model is available for HPC terminologies, we trained BERT-based entity detection and classification model (Sect. 4.3) from scratch using HPC-NLU dataset. We evaluate the performance of predicting entities and extracting them for our trained NLU model against human-supervised and labeled HPC-NLU dataset.

The training set consists of 60,000 randomly selected queries for DL, OMB, and NAS phrases from the HPC-NLU dataset including the combinations, permutations, and synonyms. Then, we used 5 million randomly selected queries from the rest of the dataset for testing. We calculated the performance metrics by comparing the NLU output versus the human-supervised labeled HPC-NLU dataset.

Table 3. Evaluation of NLU model for entity recognition - Higher is better

Test Dataset	F1-score	Precision	Recall
HPC-NLU (5M)	0.999	0.999	0.999

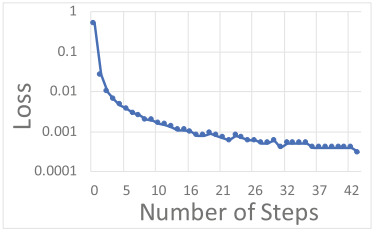

Fig. 9. Validation loss for NLU model of SAI

Table 3 shows the final test F1-score, precision, and recall on the HPC-NLU test set and achieving 99% accuracy for entity detection and classification. Figure 9 shows the validation loss of the NLU model on the HPC-NLU dataset.

6.5 Performance Evaluation of Combined ASR and NLU Models

In this experiment, ASR and NLU modules are evaluated together as a pipeline to assess the success rate of SAI for converting speech query to the classified entities. We use our trained NLU and ASR models to calculate inference accuracy. A speech test dataset of 100 queries from 4 individuals' were chosen for end-to-end inference with the following demographic: User 1 with Mandarin accents, User 2 with Middle East accents, and User 3 and 4 with American accents. The testing queries did not exist in the training queries.

As the predicted sentences of the ASR model can have different generated lengths based on the accent from the original sentences, the NLU model cannot compare entities pairwise. Thus, we designate two metrics for end-to-end testing: `Metric 1 (M1)`: if a predicted sentence has more words than the original sentence, we drop the last few words in prediction to make sure they have the same lengths and vice versa. `Metric 2 (M2)`: we first drop less important words like articles, prepositions, and grammatically wrong words inside the ASR-generated query and then

Table 4. Word Error Rate and inference accuracy for ASR+NLU pipeline of SAI on 4 users where the models were not trained on 2 users - Lower is better

Metric	User 1	User 2	User 3	User 4	**Average**
WER	10.3	8.6	8.3	4.9	**8.03**
Accuracy M2	0.97	0.90	0.80	0.95	**0.907**
Accuracy M1	0.84	0.81	0.83	0.92	**0.849**

repeat the process in metric 1. Table 4 shows the results of end-to-end inference. This end-to-end result shows the practicability of our design because ASR and NLU models have never trained with recordings from User 1 and 4 but still yield 96.8% and 90.7% test accuracy. This implies flexibility of the end-to-end model for recognizing new users' voices.

6.6 Overhead Analysis of SAI

In this experiment, we evaluate the overhead of our full pipeline deployed as a passenger app: from user speech/text input to submitting a job based on the user input. The interactive application performance would be the same or better. Since different packages available through Spack have varying installation times, we skip the overhead of package installation and job execution. In subsequent requests involving the same sets of software, this overhead won't be observed since the software is already installed.

Figure 10 showcases the average end-to-end duration to process speech and text queries of varying lengths in SAI end-to-end pipeline. In general, it can be seen that the time taken to process speech increases with an increase in the number of words in the query. This is expected as the ASR model takes an input of the varying size and hence bigger inputs take more time. The time taken to process a text query is more or less constant as the input size of the NLU model is fixed.

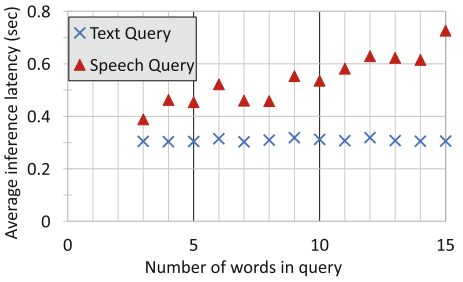

Fig. 10. End-to-end latency evaluation of SAI for 13 different queries on passenger app on speech and text queries consisting of different numbers of words - each data point is an average of 200 iterations

6.7 Overhead Analysis of Scaling Passenger App Users

As mentioned in Sect. 4.7, resources are shared among users of SAI when deployed as a passenger app. This experiment evaluates the end-to-end overhead of SAI when multiple users interact with it at the same time. To do this we use selenium with the chrome web driver to simulate different amounts of users using the SAI passenger app at once for text and speech queries. We use a barrier to ensure the concurrency of users' requests for each iteration. The test uses a text/speech query of 8 words

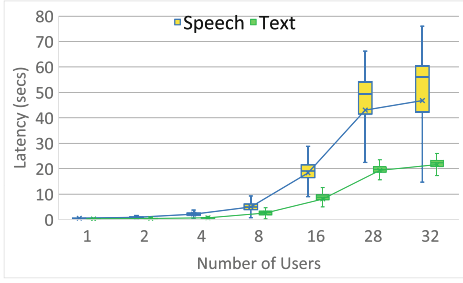

Fig. 11. Boxplot comparison of end-to-end latency of SAI passenger deployment as a varying number of concurrent users utilizing SAI for both speech and text - the host node is equipped with BDW 28 cores CPU

for 200/100 iterations per user. Figure 11 shows the box plot of scaling the users from 1 to 32 concurrent users utilizing SAI as a passenger app. We observe that as the number of users scales up both the average latency and the variance in latency scale up for both speech and text with speech increasing at a greater rate. Moreover, the performance of the login node hosting the passenger app degrades significantly. The increased latency and significant jitters of multi-user passenger apps motivate us to develop SAI as an interactive app to ensure a smooth user experience with lower latency.

6.8 Analysis of SAI Interactive App on Different Architectures

The performance degradation of SAI during scaling up the number of concurrent users motivated us to develop and evaluate SAI as an interactive application to ensure exclusive resources. The user selects the partition/architecture to run SAI. In this experiment, we evaluate the breakdown and the total latency of SAI's both deployment on different architectures for the same 8-word text and speech query running 100 and 400 iterations for speech and text respectively. The passenger test was conducted during the winter break and as many users were not using the system hence, shows the best scenario of the passenger case. The K80 GPU node did not support ASR inference. Table 5 summarizes the median end-to-end latency of total time and breakdown of latency across SAI's sub-components. We observe the latency of ASR and NLU modules decreases when inference happened on the V100 GPU node and overall the total latency is lower than the passenger deployment.

Table 5. Total latency and its breakdown for the deployment of SAI on different 1-node architectures as Interactive and Passenger app inside OnDemand - Numbers show the median of running 8-word speech/text query for 100/400 iterations respectively.

Architecture /Model	Deployment type	Total latency	ASR module	NLU module	KG module
BDW speech	Interactive	0.4919	0.23865	0.02275	0.22655
	Passenger	0.50245	0.2366	0.0217	0.2274
BDW text	Interactive	0.2665	N/A	0.0227	0.24335
	Passenger	0.27125	N/A	0.0218	0.24795
SKX speech	Interactive	0.44085	0.24105	0.0174	0.1754
SKX text		0.22095	N/A	0.0242	0.19585
V100 speech		0.40735	0.16585	0.0172	0.224
V100 text		0.2664	N/A	0.0225	0.2433
K80 text		0.2676	N/A	0.0225	0.2448

7 Discussion

7.1 Security and Authentication

SAI leverages Open OnDemand's user authentication and access privileges validation features and uses "spack verify" command, as shown in Fig. 7, to confirm the integrity of all installed binaries at any time after installation, ensuring that no files have been tampered with or modified. This added layer of security enhances user trust and the reliability of the installed packages.

7.2 Handling Ambiguous Queries in SAI

We discuss the limitations of SAI and the level of ambiguity SAI handles. We seek to see at what point SAI will not understand the user and how we handle those cases. Our developed dataset is limited to popular HPC/DL phrases hence, SAI does not understand all existing HPC synonyms or all available DL models. The HPC-ASR and HPC-NLU dataset contains synonyms and different combinations of the phrases but is limited. We have trained ASR model with 20 volunteer individuals targeting diverse dialect, but our HPC-ASR dataset is still limited. This limitation may result in SAI predicting wrong text output. To address this, SAI displays the transcript in the input text field and allows the user to correct mistakes in speech recognition if there are any. The users can switch between text and speech to resolve any discrepancies on speech recognition during conversation.

SAI shows the internal workflow (Sect. 5) enabling the user to see the parameters and packages and shows the default values. SAI always checks user argument versus the allowed range in the corresponding KG and confirms it with the user. In case anything is missing, SAI provides feedback by asking questions. For example, user can say "train resnet" and as SAI checks the related KG, it inquiries the user for an image dataset and number of nodes/processes as requirements. Currently, SAI does not support directly querying the KG. For example, users cannot ask "what are the datasets for DL image processing?" Also, SAI does not give the option to users to update the KGs.

7.3 Trade-offs for Converting Speech to Entities

There are two ways to convert speech to entities: 1) ASR followed by entity detection and classification, and 2) direct speech to entity detection and classification. Our design uses the first approach, which first converts audio to text then uses NLU to detect and classify entities. We chose this approach for several reasons: first, a speech to entity model requires a large corpus of labeled HPC speech datasets for optimal accuracy; second, since SAI supports both speech and text input, creating separate datasets would be necessary (one for NLU and one for ASR); Third, pre-trained DL models for similar tasks are not available. Fourth, this approach allows for easy integration of new software, only requiring a few minutes of audio recordings containing its terminologies.

7.4 Portability for New Software and Systems

To extend SAI support to a new HPC software, SAI-O ontology can be used to capture the relationships of a new application to be added to SAI. We represent these relationships by using KGs, which capture the connections between software, data, models/algorithms, systems, and arguments.

Adding a new application to SAI requires two steps: (1) creating a KG for the application using the SAI-O ontology and the supported relationships (Table 1), and (2) adding application-specific terms to the HPC-ASR dataset. SAI will provide easy-to-use scripts for fine-tuning the ASR model on new audio samples, enabling support for new terminologies in ASR model. The KG Inference module selects the appropriate KG using "spack verify", allowing us to reuse the general query manager and simplify the addition of new applications. We have trained the NLU module on a large dataset and therefore, it can detect entities and classify them into broad categories based on the sentence structure. In rare cases, the performance of the NLU module may degrade due to new terminologies, but SAI provides an easy-to-use dataset generator script to generate new text commands based on models/algorithms, datasets, software, arguments, and systems to fine-tune the DL model and improve performance for the new application. Figure 7 shows a setting interface where users can upload their customized trained DL models to be used for SAI.

The modularity of our design allows the KG to be ported to multiple systems by updating a template (Sect. 9) with the new platform's system information. Integrating SAI with Open OnDemand makes it even easier to port to new system architectures, as many XSEDE/ACCESS systems use OnDemand. The KG's system portion is the same for all applications on a system, simplifying the deployment of SAI on a new HPC system.

8 Related Work

Our previous work [14] introduced a new conversational AI interface for HPC profiling tools like OSU INAM [15], with a focus on extracting performance-related terminologies, intents, and slots for HPC tools and scope of profiling tools. In this paper, we expand upon our previous work by capturing a broader range of HPC runtime terminologies, and by incorporating an NLU entity recognition model to process user inputs. Additionally, we introduce new features to enhance the interface's usability and effectiveness, such as integration with OnDemand, a software installer component, and job submission capabilities. Several studies [20,26] exist in literature that uses an end-to-end based approach to convert the voice to intent and slots, combining ASR and NLU into one model. The trade-off is discussed in Sect. 7.3 and maintaining and updating a list intent and slots causes the KG query module not to be portable. Another approach is to combine ASR and NLU models to understand the context of speech samples. The state-of-the-art ASR models [9,27] have been proposed in the literature that provides good performance for publicly available datasets and common words found in day-to-day conversation. However, we need to fine-tune these ASR models to

recognize technical terms found in computer science and HPC. Similarly, NLU models [7,8,10,23] are trained for publicly available datasets. Hence, to develop a system for HPC software installation and usage tool, we need to generate our own dataset and retrain models from scratch to get better accuracy. To the best of our knowledge, this is the first work that develops a conversational AI-based interface for HPC software installation and execution.

9 Future Work

As part of future work, we plan to simplify the process of creating KGs for HPC software by providing easy-to-use templates to create a KG for a given application and collaborating with multiple HPC centers to identify common applications and make corresponding templates available to users. We will provide examples of existing applications to assist in filling the template. To create a new application's KG, users need to provide the model/application, data, arguments, and dependencies to the KG template. As the repository of model commons for the templates grows, users can update the templates with new features and system information and contribute to the repository, ultimately saving time and effort of generating KGs's template for common applications. Furthermore, we plan to expand the SAI-O ontology to capture a broader range of HPC applications and runtimes, including those with complex inter-dependencies and unique configurations. Finally, we plan to release SAI and our solutions.

10 Conclusion

In this paper, we proposed - SAI, a Conversational AI-Enabled Interface for science gateways in HPC. We created an HPC speech and text dataset to train Automatic Speech Recognition and Entity detection and classification model to understand the input. By defining a new ontology, called `SAI-O`, we provided a general approach for any HPC application by using knowledge graphs to check and validate the task given by the user. This allowed us to get default values for optional arguments and design a conversational interface to get the required arguments for running the application. We demonstrated the capability of the proposed design by supporting three different HPC applications: 1) OSU Microbenchmarks, 2) Distributed DNN training, and 3) NAS parallel benchmarks. Finally, we integrated SAI in Open OnDemand and deployed it on real HPC systems. We also evaluated its performance and functionality. To the best of our knowledge, this is the first attempt in the HPC field to enhance the user experience by designing a AI-powered speech-assisted interface. Early users have shown interest and found SAI features very useful to onboard domain scientists to HPC.

References

1. TIMIT acoustic-phonetic continuous speech corpus. https://hdl.handle.net/11272.1/AB2/SWVENO
2. SPARQL query language (2020). https://www.w3.org/TR/sparql11-query/. Accessed 17 April 2023
3. Voicebot research (2020). https://tinyurl.com/4kw4bmz7
4. The future of conversational AI (2021). https://tinyurl.com/2dzxe2w8
5. Open onDemand (2022). https://osc.github.io/ood-documentation/latest/#
6. The impact of voice assistants (2022). https://tinyurl.com/mrx36afk
7. Hosseini-Asl, E., McCann, B., Wu, C.S., Yavuz, S., Socher, R.: A simple language model for task-oriented dialogue (2020). CoRR abs/2005.00796. https://arxiv.org/abs/2005.00796
8. Wen, T.H., Gasic, G., Mrksic, N.S., Vandyke, D., Young, S.J.: A network-based end-to-end trainable task-oriented dialogue system (2016). CoRR abs/1604.04562, http://arxiv.org/abs/1604.04562
9. Baevski, A., Zhou, H., Mohamed, A., Auli, M.: Wav2vec 2.0: a framework for self-supervised learning of speech representations (2020). https://arxiv.org/abs/2006.11477
10. Castellucci, G., Bellomaria, V., Favalli, A., Romagnoli, R.: Multi-lingual intent detection and slot filling in a joint bert-based model (2019). https://arxiv.org/abs/1907.02884
11. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding (2018). CoRR abs/1810.04805, http://arxiv.org/abs/1810.04805
12. Goasduff, L.: Chatbots will appeal to modern workers (2019). https://www.gartner.com/smarterwithgartner/chatbots-will-appeal-to-modern-workers
13. Hauptmann, A., Rudnicky, A.: A comparison of speech and typed input (1990). https://doi.org/10.3115/116580.116652
14. Kousha, P., et al.: "Hey CAI" - conversational AI enabled user interface for HPC tools. In: Varbanescu, A.L., Bhatele, A., Luszczek, P., Marc, B. (eds.) High Perform. Comput., pp. 87–108. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-07312-0_5
15. Kousha, P., et al.: INAM: cross-stack profiling and analysis of communication in MPI-based applications. In: Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437359.3465582
16. Kudo, T., Richardson, J.: Sentencepiece: a simple and language independent subword tokenizer and detokenizer for neural text processing (2018). arXiv preprint arXiv:1808.06226
17. Liao, C., Lin, P.H., Verma G., Vanderbruggen, T., Emani, M.: Hpc ontology: towards a unified ontology for managing training datasets and AI models for high-performance computing. In: 2021 IEEE/ACM Workshop on MLHPC, pp. 69–80 (2021). https://doi.org/10.1109/MLHPC54614.2021.00012
18. National Geographic: LiDAR and Archaeology. https://education.nationalgeographic.org/resource/lidar-and-archaeology
19. OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
20. Palogiannidi, E., Gkinis, I., Mastrapas, G., Mizera, P., Stafylakis, T.: End-to-end architectures for ASR-free spoken language understanding. In: (ICASSP), pp. 7974–7978 (2020). https://doi.org/10.1109/ICASSP40776.2020.9054314

21. Panayotov, V., Chen, G., Povey, D., Khudanpur, S.: Librispeech: an ASR corpus based on public domain audio books. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5206–5210. IEEE (2015)
22. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library (2019)
23. Qin, L., Che, W., Li, Y., Wen, H., Liu, T.: A stack-propagation framework with token-level intent detection for spoken language understanding (2019). arXiv preprint arXiv:1909.02188
24. Rothwell, B., Sgambati, M., Evans, G. Biggs, B., Anderson, M.: Quantifying the impact of advanced web platforms on high performance computing usage, PEARC'22. ACM (2022). https://doi.org/10.1145/3491418.3530758
25. Schmidt, A.: The rise of conversational interfaces and their impact on business (2019). https://tinyurl.com/45ppfz9t
26. Serdyuk, D., Wang, Y., Fuegen, C., Kumar, A., Liu, B., Bengio, Y.: Towards end-to-end spoken language understanding (2018). CoRR abs/1802.08395, http://arxiv.org/abs/1802.08395
27. Wang, C., Tang, Y., Ma, X., Wu, A., Okhonko, D., Pino, J.: Fairseq s2t: fast speech-to-text modeling with fairseq (2020). https://arxiv.org/abs/2010.05171