Model Customization for Code Creation with Red Hat OpenShift AI on Dell AI Optimized Infrastructure

How training and tuning an AI model can help your enterprise's evolution to better code

July 2024

H20100

Technical White Paper

Abstract

This **Dell Reference Design** describes the model customization and inferencing scenarios on Dell Technologies' AI optimized infrastructure in conjunction with Red Hat OpenShift AI. This solution offers developers, increased productivity and code quality by training and tuning a large language model. It provides insights into hardware infrastructure requirements, OpenShift AI software components, the dataset selection, and the LoRA customization techniques.

Dell Technologies AI Solutions

Dell
Reference Design

DELL Technologies

Copyright

Contents

Introduction

Overview

AI is becoming integrated into all aspects of our lives and having a real impact on almost all the ways we conduct business and provide services ranging from chat to code development. As businesses continue to define what their AI journey will look like, most are aware that they must start such initiatives to stay relevant.

One of the limitations of the base large language models is their generic behavior which needs customization and adaptation to a specific business use case using customization techniques. Also, LLMs, once trained and generated, do not have access to information beyond the date that they were trained.

There are numerous techniques that can be used to overcome these limitations, one of them is retrieval augmented generation (RAG). RAG extends the functionality of the LLMs by retrieving facts from an external knowledge base hosted using a vector database such as Redis.

Another technique that can be used to fine-tune the model is using a specific dataset. This approach modifies the parameters of the base model making them customized to a business use case such as code development, chatbot, digital assistant, or language translator and transcriber. Fine-tuning aims to maintain the original capabilities of a pretrained model while adapting it to suit more specialized use cases.

Before deciding which approach to use, one should consider the pros and cons of training your base model, fine-tuning an existing model and RAG:

- Training an LLM from scratch requires compute and GPU resources and expertise that are scarce. However, this approach gives you the most control over the final model and its output.
- Fine-tuning an existing model is something that most can do but it requires a significant amount of time. In addition, any updates to the data used for training require the fine-tuning process to be repeated. However, a fine-tuned model will probably have a lower OPEX because there is no need for additional, RAG related, HW, and SW. This is a good option when your source data will not change frequently, and cost is a primary concern.
- LoRA is an efficient fine-tuning method where instead of fine-tuning all the weights that constitute the weight matrix of the pretrained LLM, it optimizes rank decomposition matrices of the dense layers to change during adaptation. These matrices constitute the LoRA adapter. This fine-tuned adapter is then merged with the pretrained model and used for inferencing. The number of parameters is determined by the rank and shape of the original weights. In practice, trainable parameters vary as low as 0.1% to 1% of all the parameters. As the number of parameters needing fine-tuning decreases, the size of gradients and optimizer states attached to them decrease accordingly. Thus, the overall size of the loaded model reduces.
- RAG requires the least amount of fine-tuning and allows the data in the vector DB to be updated continuously. Therefore, RAG is better when data is updated frequently.

This technical white paper provides an example of fine-tuning an LLM running on Dell Technologies AI optimized infrastructure demonstrating a robust developer-friendly ecosystem provided by Red Hat OpenShift AI. The use case consists of a scenario where administrators and operators can manage the life cycle of a large language model from the model customization using distributed fine-tuning to model serving and inferencing to add business value. An ecosystem solution is defined which is developer-friendly, providing a centralized hub where supported software components are available greatly increasing the downstream productivity of large language models.

About this document

The purpose of this document is to present a reference design for a system to perform automated code generation. The system is based on Dell PowerEdge servers with NVIDIA GPUs and Red Hat OpenShift software. The system will leverage generative AI components and processes.

Note: The contents of this document are valid for the described software and hardware versions. For information about updated configurations for newer software and hardware versions, contact your Dell Technologies sales representative.

Audience

This design guide is intended for AI solution architects, data scientists, data engineers, IT infrastructure managers, and IT personnel who are interested in, or considering, implementing AI and ML deployments.

Solution Overview

Business challenges

In today's high stakes business environments, fast, agile, and accurate development of applications for internal or external use is imperative to stay relevant and deliver wanted outcomes. The impediments to high-performing development teams are not new and tools in the past have tried to address the roadblocks with varying degrees of success.

Routine and repetitive tasks can degrade the productivity of the software development process by increasing the time and effort spent on coding and limiting time better spent on higher-level system design and logic.

Understanding complex code bases can be difficult even for seasoned developers. In such environments, it can be difficult to identify, debug, and correct code errors early in the feature development and reduce technical debt.

Beyond the code, the skill set and experience of new team members may be insufficient to keep productivity and quality high. Training can be a long-term answer here but can also be expensive in terms of short-term productivity and time consumed.

Addressing these shortcomings across teams and processes can enable businesses to generate accurate and efficient code and maintain their development schedules even in the face of resource constraints.

Solution approach

The proposed approach to solving these business challenges includes hardware and software components as well as an overall process for continuous improvement.

Major process steps include training a base coding large language model on additional data then deploying the new model for testing and use as a code generation and explanation system.

The first step is choosing the model. Large language models such as those in the Code Llama family are desirable since they have been trained on vast libraries of code for infilling and new code generation purposes. The most widely used coding scenarios which follow standards have higher weightage when it comes to tune parameters in the model. This results in code recommendations which follow widely accepted industry standards.

The higher the number of tuned parameters the LLM consists of, the higher the accuracy it provides. Hence, for the best accuracy, the Code Llama 70b parameter version is recommended. However, if GPU and compute resources are limited and faster results are wanted at the cost of accuracy, 7b, 13b, or 34b models can be used.

Note: As the state of the art in large language models changes over time, part of the implementation should have capabilities to continually evaluate new models and technology to drive improvements in output.

For the hardware in this solution, a scalable, distributed, multi-tier multi-purpose configuration is proposed. This hardware division of labor includes the following major components and characteristics:

- Training CPU/GPU - worker
    - Scale up and out to increase model capacity/capability and decrease training duration
    - Distribute training workloads across GPUs and nodes
- Model storage (base/current, new merged, new LoRA adapters)
    - With LLMs and datasets ranging in size up to and beyond 1 TB, capacity planning is required to store multiple versions of large models and datasets. Storage accessibility should be at least 25Gps to drive down model load/save times and shared across training and inferencing nodes
    - Examples: PowerScale, ObjectScale, OpenShift Data Foundation (ODF)
- Inferencing - worker
    - Provides endpoints to generate predicted data based on input prompts from end users
    - Monitoring and planning are required to adapt response speed as users scale
    - Multiple techniques are available to optimize throughput including distributed inferencing
- Platform management (OpenShift, OpenShift AI) – control-plane

Red Hat OpenShift AI running on top of its OpenShift container platform provides a robust developer-friendly ecosystem when deployed on Dell Technologies AI optimized infrastructure. With this hardware and software, we can take advantage of multi-GPU training and inferencing by using newly released distributed workloads features of Red Hat OpenShift AI.

Distributed workloads provide the following benefits[1]:

- You can iterate faster and experiment more frequently because of the reduced processing time.
- You can use larger datasets, which can lead to more accurate models.
- You can use complex models that could not be trained on a single node.

[1] https://docs.redhat.com/en/documentation/red_hat_openshift_ai_self-managed/2.11/html/working_with_distributed_workloads/overview-of-distributed-workloads_distributed-workloads#overview-of-distributed-workloads_distributed-workloads

- You can submit distributed workloads at any time, and the system then schedules the distributed workload when the required resources are available.

The ongoing functions of the solution include training data and model storage and curation, training framework, and the infrastructure for testing and production inferencing. Finally, the overall continuous improvement process can be loosely defined as the following steps:

1. Download the base model and dataset
2. Use LoRA to customize and fine tune model
3. Merge LoRA adapter back to the original model and save
4. Deploy and test inferencing on the original model
5. Deploy and test inferencing on the newly trained model
6. Evaluate and compare results of inferencing

Solution design

Hardware design

An overview of the hardware environment used for training and inferencing are shown in the following diagram.

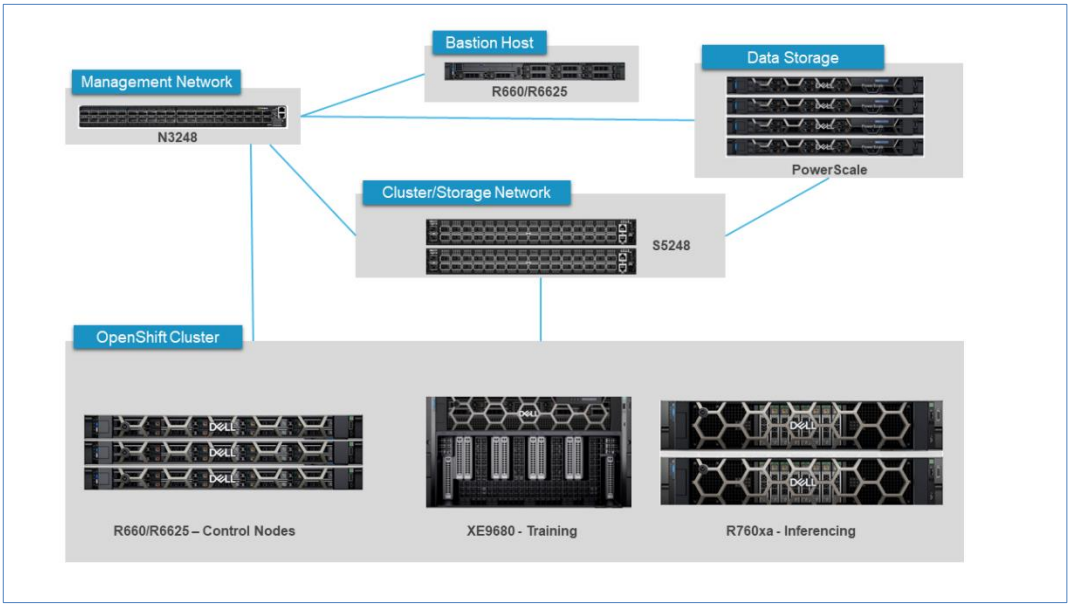

Figure 1. Hardware and networking overview diagram

Servers

The testing of the solution in this paper focused on the distributed training across GPUs and do performance and scalability testing was performed. The following table contains the equipment used. As an alternative for control nodes, Dell PowerEdge R660 servers based on Intel Xeon CPUs Intel-CPUs can be used.

Node Type	Model	CPU	RAM	GPUs
OpenShift Control Nodes	3 x R6625	2 x AMD EPYC 9354 32 Cores	768 GB	0
OpenShift Inferencing Workers	2 x R760xa	2 x Intel Xeon Gold 6454S 32 Cores	512 GB	4 x NVIDIA H100 PCIe 80 GB
OpenShift Training Worker	1 x XE9680	2 x Intel Xeon Platinum 8470 52 Cores	2048 GB	8 x NVIDIA HGX H100 SXM 80 GB

Storage

For the implementation and testing of this solution OpenShift Data Foundation was used for object storage for saving of the LoRA checkpoints during training. OpenShift Data Foundation integrates Ceph with multiple storage presentations including object storage (compatible with S3), block storage, and POSIX-compliant shared file system.

To facilitate time efficient model training/deployment and flexible model versioning, it is recommended to use fast shared storage such as a PowerScale system with the NFS CSI driver. The CSI Driver for PowerScale is part of the CSM (Container Storage Modules) open-source suite of Kubernetes storage enablers for Dell Technology (Dell) products. For more information about deployment and implementation of the CSI driver, see https://github.com/dell/csi-powerscale.

Networking

Although the inferencing and training activities described in this document use only a single node at any point in time, for multi-node AI activities, networking plays a critical role in interconnecting GPU servers to optimize the GPU fabric that enhances the ability of the environment to perform the distributed AI tasks. In addition, for all types of AI environments, loading AI models from storage and keeping temporary checkpoints in storage using frontend fabric during model training means that network component selection is a critical element of any architectural design. Dell's flagship PowerSwitch products for GenAI solutions are the Z9664F and Z9864F.

Software design

To be an effective and easy to manage training and inferencing solution the following components have been used.

Platform Software

The overall solution installation is built on top of the Red Hat OpenShift platform. With a robust AI ecosystem, OpenShift yields an enterprise-ready foundation for modern AI applications and workloads like model training and inferencing.

For information about the deployment and configuration of the Red Hat OpenShift cluster on PowerEdge servers review the latest Dell Validated Design implementation guide documents available at: https://infohub.delltechnologies.com/en-us/t/red-hat-openshift-container-platform/

With simple installation using the integrated OperatorHub, OpenShift supported and certified operators provide additional key software resource layers and programmatic access to hardware resources. Important operators installed in this solution are the NVIDIA GPU operator, OpenShift Data Foundation, and Red Hat OpenShift AI.

AI or Solution Software

Red Hat OpenShift AI enables AI adoption with an integrated MLOps platform for building, tuning, deploying, and monitoring AI-enabled applications and models at scale.

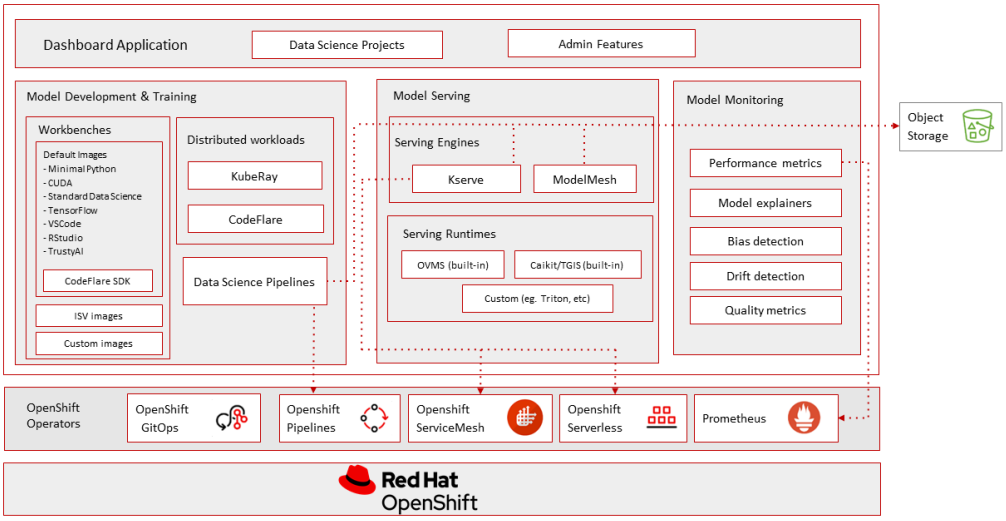

Figure 2. Red Hat OpenShift AI Architecture

The following software component versions were used and are discussed in further detail below.

Software Component	Version
Red Hat OpenShift	4.15.5
Red Hat OpenShift AI	2.11
NVIDIA GPU Operator	23.9.2
Distributed workloads components (managed by Red Hat OpenShift AI)[2]	CodeFlare KServe Ray Kueue Workbenches Dashboard
Workbench Notebook Image	Red Hat Standard Data Science Notebook Image Version 2024.1 Python 3.9
LLM	meta-llama/CodeLlama-70b
Dataset	cognitivecomputations/dolphin-coder
vLLM	0.5.0-post1
Visual Studio Code	1.90.0
Continue Plugin	0.8.43

[2] See https://access.redhat.com/articles/rhoai-supported-configs for more information about dependencies and GA status.

Jupyter notebooks enabled by Red Hat OpenShift AI provide a sandbox for development and execution of Python code related to model training.

From a notebook, the CodeFlare framework enables direct workload configuration (for example Ray cluster creation) and simplifies the task orchestration and monitoring. It also offers seamless integration for automated resource scaling and optimal node utilization with advanced GPU support.

The Ray cluster provides a flexible and scalable distributed set of head and worker pods on which the Python training code is run.

As described in this GitHub page, DeepSpeed, an open-source deep learning optimization library for PyTorch, is used for the basis of the distributed fine-training code for the Code Llama 70b model using the LoRA method.

Finally, for fast and easy-to-use inference and serving of base and trained models, the vLLM library is used. For the end user experience, Visual Studio Code is the IDE using the Continue plugin to interface with the inference server to generate and explain code.

Large Language Models

Code Llama is a collection of pretrained and fine-tuned generative text models ranging in scale from 7 billion to 34 billion parameters designed for general code synthesis and understanding. This solution focuses on Code Llama and fine-tuning using LoRA technique with a dataset available from Hugging Face as an example for developers. The developers can use these guidelines to customize these models according to their datasets based on their own software repositories.

Implementation guidance

Event and monitoring telemetry data is available across the components in the solution. It is recommended to leverage this data to understand the utilization of hardware resources and to drive changes to increase efficiencies.

Within Red Hat OpenShift, The NVIDIA GPU operator exposes GPU telemetry for Prometheus by using the NVIDIA DCGM Exporter. These metrics can be visualized using a monitoring dashboard in Grafana.

Red Hat OpenShift AI distributed workloads metrics can be easily viewed and monitored within the OpenShift AI web UI. The Ray cluster dashboard can be used to monitor the status and progress of training jobs within the environment and TensorBoard (TensorFlow dashboard) can show model experiment metrics such as loss and accuracy to help with evaluation.

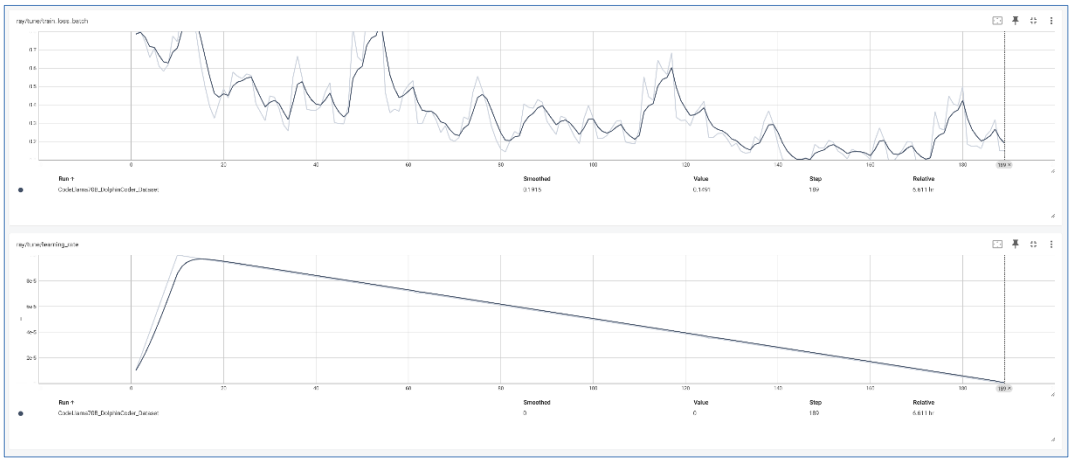

Figure 3. TensorBoard showing training loss and learning rate

When using large models and datasets, storage capacity and performance are other factors that should be monitored and considered in the environment. Efforts can be made to use model caching or local storage to decrease the time spent copying model files across network endpoints.

Security considerations

Deploy and use the hardware and software recommended in this solution securely. Follow recommendations from Dell Technologies, and the other vendors cited in this reference design. For additional information, refer to the Dell Technologies Security and Trust Center at https://www.dell.com/en-us/dt/about-us/security-and-trust-center/index.htm.

Results or findings

Training

With the hardware and software platform available, the first training experiment run can be performed.

The steps for distributed fine-tuning training follow the example from this repository: https://github.com/opendatahub-io/distributed-workloads/tree/main/examples/ray-finetune-llm-deepspeed

The first step is to clone the repository which includes the notebook and necessary files and customize to the environment. The create_dataset.py python program can be modified to download a dataset (for example cognitivecomputations/dolphin-coder) and split it into the appropriate directories for the training (when the main notebook is run). By default, the training and testing jsonl files will be “./data/train.jsonl” and “./data/test.jsonl”

When the dataset is ready, launch the “ray_finetune_llm_deepspeed.ipynb” notebook file.

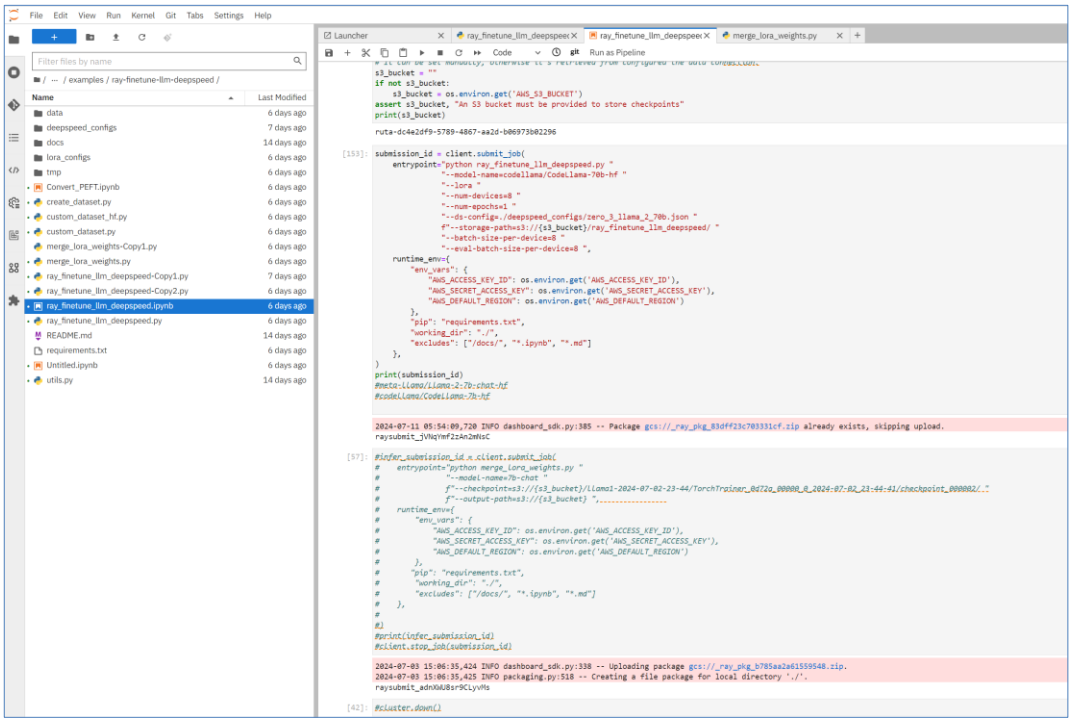

Figure 4. Fine-tuning Jupyter Notebook

After importing python modules for the notebook, creating the dataset directory, authenticating the CloudFlare SDK, the next step is to define the Ray cluster parameters.

To use all eight GPUs in the PowerEdge XE9680, the Ray cluster was configured with the following settings:

```
cluster = Cluster(ClusterConfiguration(
    name='codellama',
    namespace='drd',
    num_workers=7,
    min_cpus=12,
    max_cpus=12,
    head_cpus=12,
    min_memory=200,
    max_memory=200,
    head_memory=200,
    head_gpus=1,
    num_gpus=1,
    image="quay.io/rhoai/ray:2.23.0-py39-cu121-torch",
))
```

The next few calls in the Jupyter Notebook starts up the Ray cluster and shows the status. When ready the following Ray job can be submitted to the cluster for distribution across the eight pods in the cluster.

```
submission_id = client.submit_job(
    entrypoint="python ray_finetune_llm_deepspeed.py "
                "--model-name=codellama/CodeLlama-70b-hf "
                "--lora "
                "--num-devices=8 "
                "--num-epochs=1 "
                "--ds-config=./deepspeed_configs/zero_3_llama_2_70b.json "
                f"--storage-path=s3://{s3_bucket}/ray_finetune_llm_deepspeed/ "
                "--batch-size-per-device=8 "
                "--eval-batch-size-per-device=8 ",
    runtime_env={
        "env_vars": {
            "AWS_ACCESS_KEY_ID": os.environ.get('AWS_ACCESS_KEY_ID'),
            "AWS_SECRET_ACCESS_KEY": os.environ.get('AWS_SECRET_ACCESS_KEY'),
            "AWS_DEFAULT_REGION": os.environ.get('AWS_DEFAULT_REGION')
        },
        "pip": "requirements.txt",
        "working_dir": "./",
        "excludes": ["/docs/", "*.ipynb", "*.md"]
    },
)
```

Note: The default job parameters are defined download the model from Hugging Face. The best practice would be to enable caching of the model locally for use in additional training run experiments and across the ephemeral Ray clusters to save network bandwidth and time.

The Ray cluster dashboard showing processing results can be viewed, and individual job status monitored as shown in Figure 5.

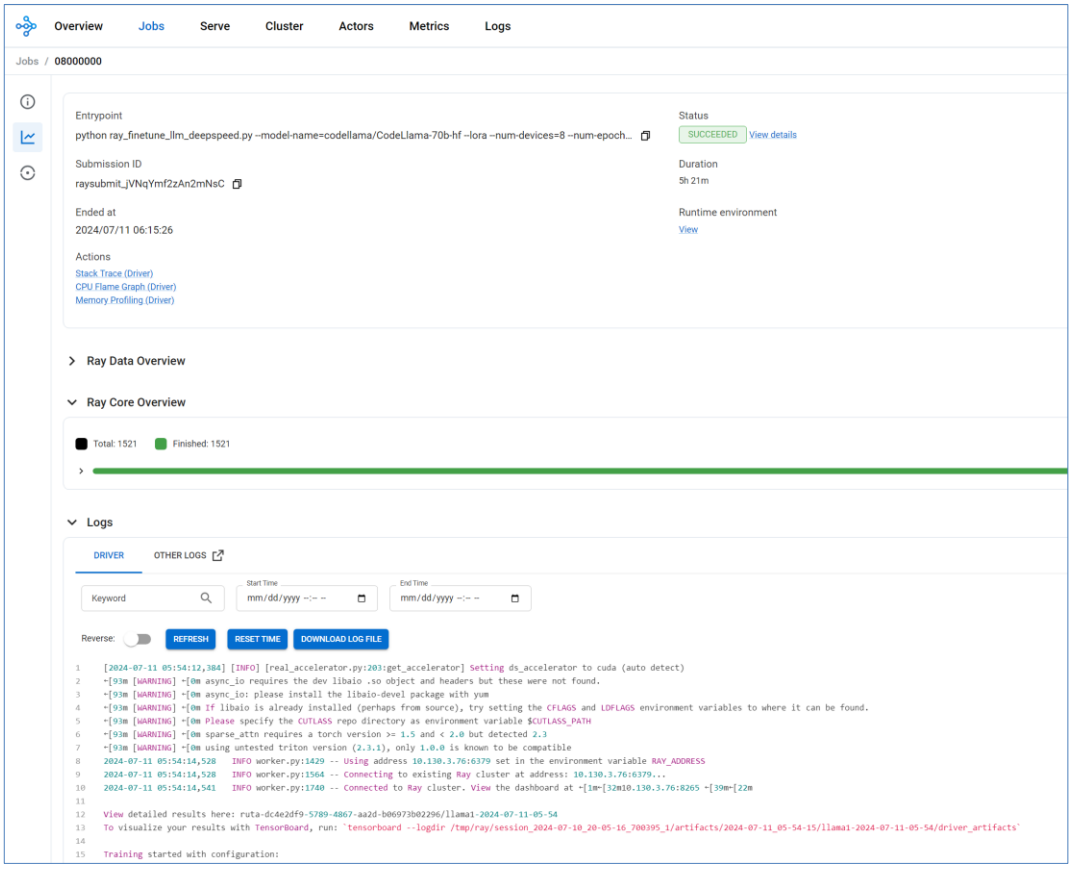

Figure 5. Ray cluster job status page

When the training is finished the logs will show where the results are stored including the best checkpoint.

The merge_lora_weights.py python script, from the Ray project on GitHub, was used to merge the checkpointed LoRA weights back to the base model used in training.

Inferencing

With the model now tuned an inference server can be configured with the model to see the results. In the hardware configuration there are two PowerEdge R760 systems each with 4 * H100 GPUs which can be used as inference servers, one for the base model and one for the fine-tuned model. As new models are trained, evaluated, and promoted, it would be possible to set them up in a development/production pair.

vLLM can be used as the inference serving runtime leveraging KServe (integrated to Red Hat OpenShift AI) to startup and scale the inferencing as necessary and appropriate. Although the vLLM serving runtime is now part of the OpenShift AI further customization is possible as shown in the YAML configuration files available here:

https://github.com/openshift-psap/inference/tree/api-server/language/llama2-70b/api-endpoint-artifacts

An example of such customization is to enable a local node PVC for model storage. During the refining of the training and, the overall process local storage can be leveraged as a fast medium to reduce load times for large models.

The KServe user guide has examples of other storage types that can be used with vLLM serving runtime instances.

When the vLLM inference server instance is completely up and running KServe provides a OpenShift route for simple access to the provided API endpoints (for example https://codellama-70b-loraft-drd.apps.eclipse.ai.lab)

A simple test to ensure the inference server is working is to query the “/v1/models” endpoint using a web browser, a command line tool like cURL or some basic python code which will return text similar to the following:


```
"object":"list","data":[{"id":"/mnt/models/CodeLlama-70b-
finetuned","object":"model","created":1721847460,"owned_by":"vllm","root":"/mnt/mod
els/CodeLlama-70b-
finetuned","parent":null,"max_model_len":2048,"permission":[{"id":"modelperm-
fae45d29b08d4ce2ad1b6a4f74a5fdc3","object":"model_permission","created":1721847460,
"allow_create_engine":false,"allow_sampling":true,"allow_logprobs":true,"allow_sear
ch_indices":false,"allow_view":true,"allow_fine_tuning":false,"organization":"*","g
roup":null,"is_blocking":false}]}]}
```

Visual Studio Code + Continue Plugin

To test the efficacy of the newly trained model, Visual Studio Code is used to simulate a real-world usage example of code generation/explanation. Since VS Code does not have a native interface to LLM inference services the “Continue” plugin is used and configured to submit queries to an inference service serving our original and then newly trained model.

The installation of the Continue plugin is easy accomplished from the VS Code marketplace. The configuration is simple and only some basic metadata about the model being served, and the inference endpoint is required.

If resources are available to run multiple simultaneous inference servers, then additional models can be defined in the Continue configuration for easy switching and testing.

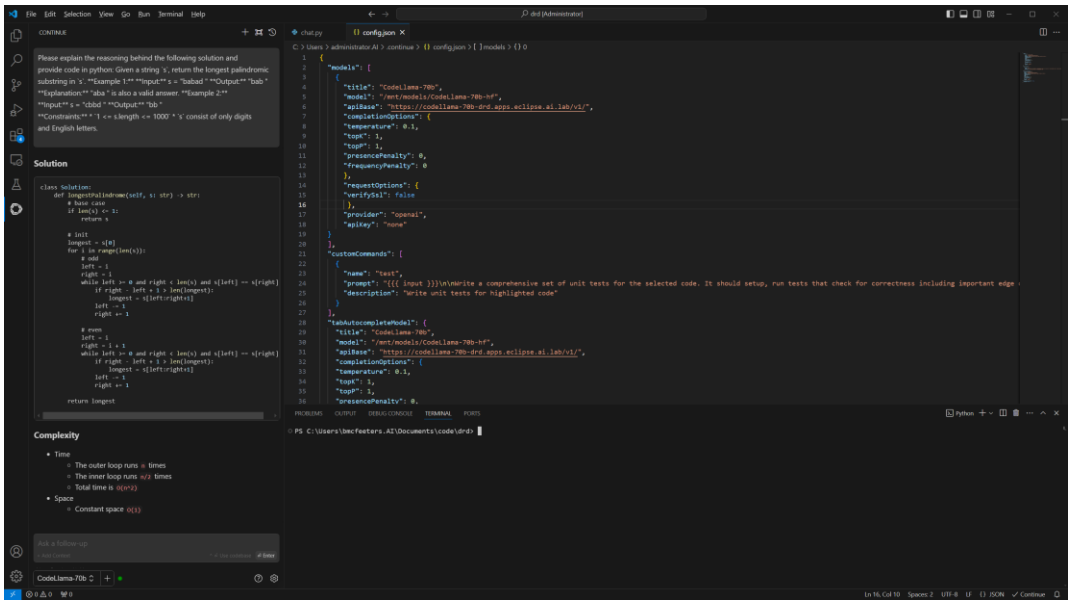

Figure 6. Base model inferencing with VS Code

As seen in Figure 6, the original Code Llama 70b model responds to our chat prompt asking about finding the largest palindrome inside a string using the Python programming language.

A solution is generated in Python as request and the model also returns information about the time and space complexity of the resulting code.

When asking the same question with the model fine-tuned with LoRA, we can observe a different and qualitatively[3] better result in Figure 7. The model returns the explanation section we had requested along with pseudocode, actual Python code, and complexity characteristics as before.

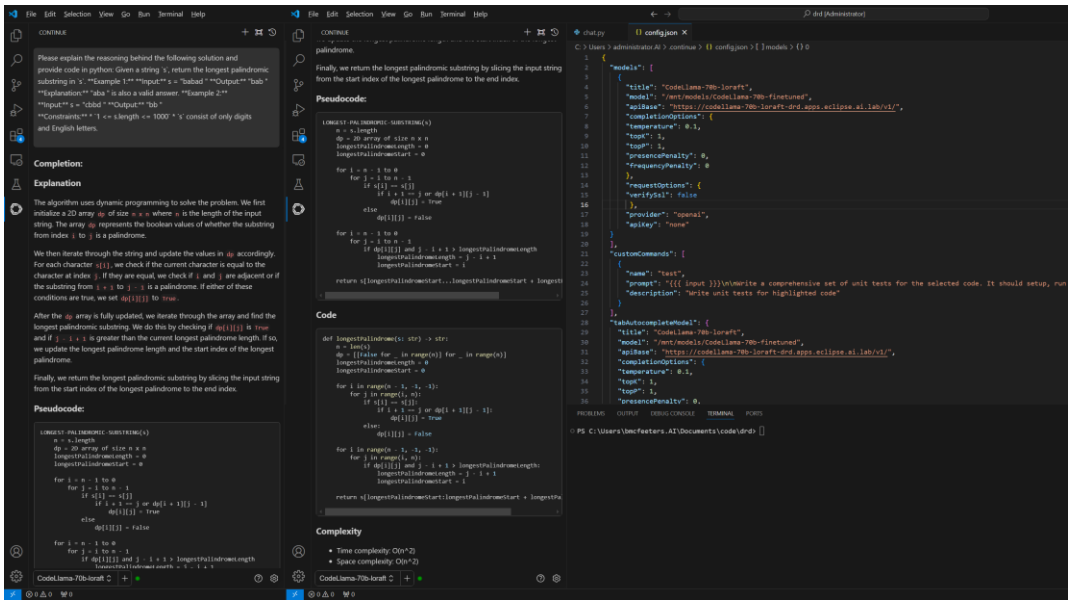

Figure 7. Inference results in VS Code with fine-tuned model

[3] For more information about the comparison see this similar experiment: https://aws.amazon.com/blogs/machine-learning/fine-tune-code-llama-on-amazon-sagemaker-jumpstart/

Conclusion

We have outlined a highly performant code large language model training and inferencing system using scalable components for additional performance where/when necessary.

The testing performed shows that training an LLM code on additional datasets can provide valuable additional information and accuracy when used during development activities.

The testing only scratched the surface of how an LLM can be trained/tuned and optimized for specific tasks within a specific company. Different models and datasets can be used to continue to refine the output of the system to yield even more amazing outcomes.

We value your feedback

Dell Technologies and the authors of this document welcome your feedback on this document and the information that it contains. Contact the Dell Technologies Solutions team by email.

References

Dell Technologies documentation

The following Dell Technologies documentation provides additional and relevant information related to this solution.

- Dell Technologies Info Hub for AI Solutions

Partner documentation

The following partner and third-party documentation provide additional and relevant information related to this solution.

- Red Hat OpenShift AI