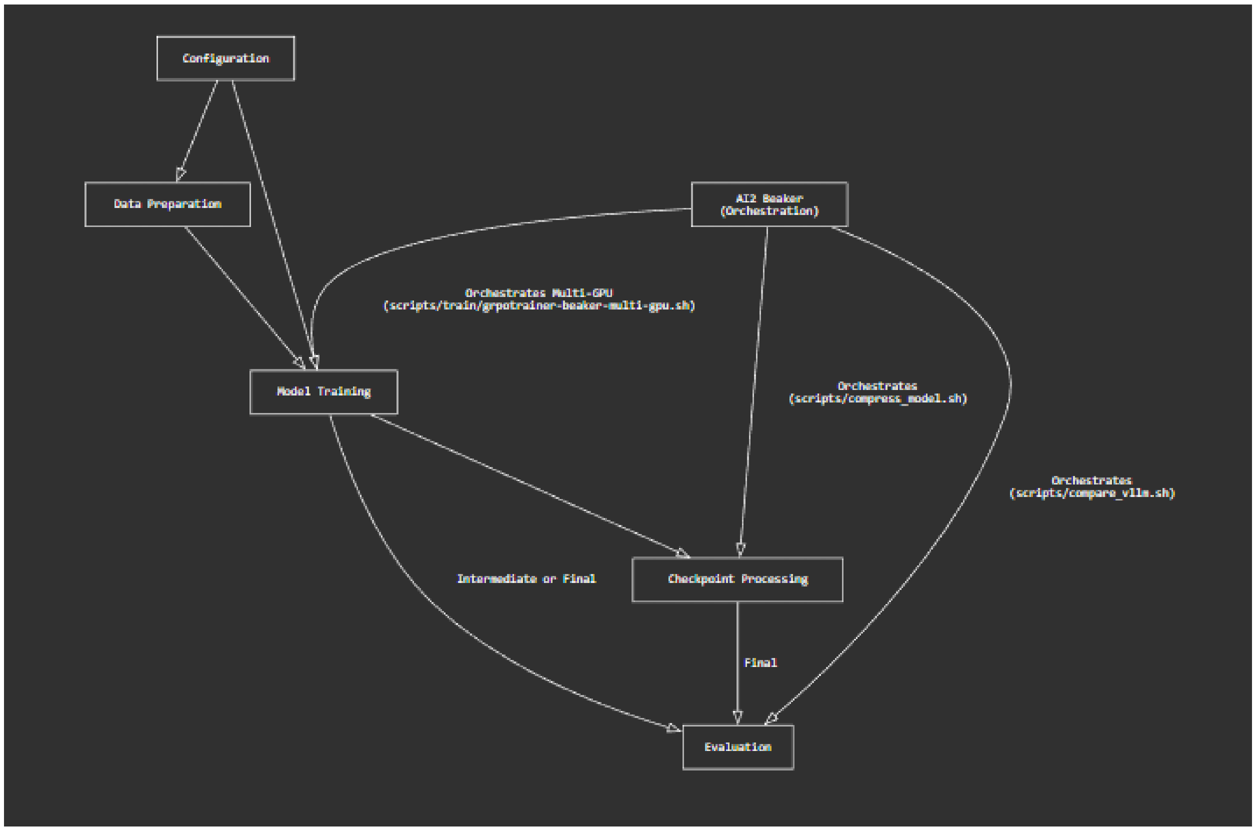

Configuration
Data Preparation
AI2 Beaker (Orchestration)
Orchestrates Multi-GPU (scripts/train/grpotrainer-beaker-multi-gpu.sh)
Model Training
Orchestrates (scripts/compress_model.sh)
Orchestrates (scripts/compare_vllm.sh)
Intermediate or Final
Checkpoint Processing
Final
Evaluation

The olmOCR project provides a comprehensive framework for developing, evaluating, and deploying optical character recognition (OCR) models. This involves structured processes for training models, optimizing their size and performance through quantization, and ensuring their reliability through rigorous validation. The training workflows support advanced techniques like Generalized Reinforcement Learning with Policy Optimization (GRPO), which refines model behavior using aggregated reward functions derived from document understanding benchmarks.

Model development begins with defining detailed configurations for the training process. These configurations, managed through Python dataclass objects in olmocr/train/config.py, specify various parameters including model architecture, dataset paths, training hyperparameters, and logging settings. This modular approach allows for flexible and reproducible experimentation. Data preparation involves loading markdown-PDF pairs using BaseMarkdownPDFDataset in olmocr/train/dataloader.py, where documents undergo a series of pipeline steps such as parsing front matter, rendering PDFs to images, extracting anchor text, and generating finetuning prompts. This ensures that the raw data is transformed into a format suitable for multimodal model training.

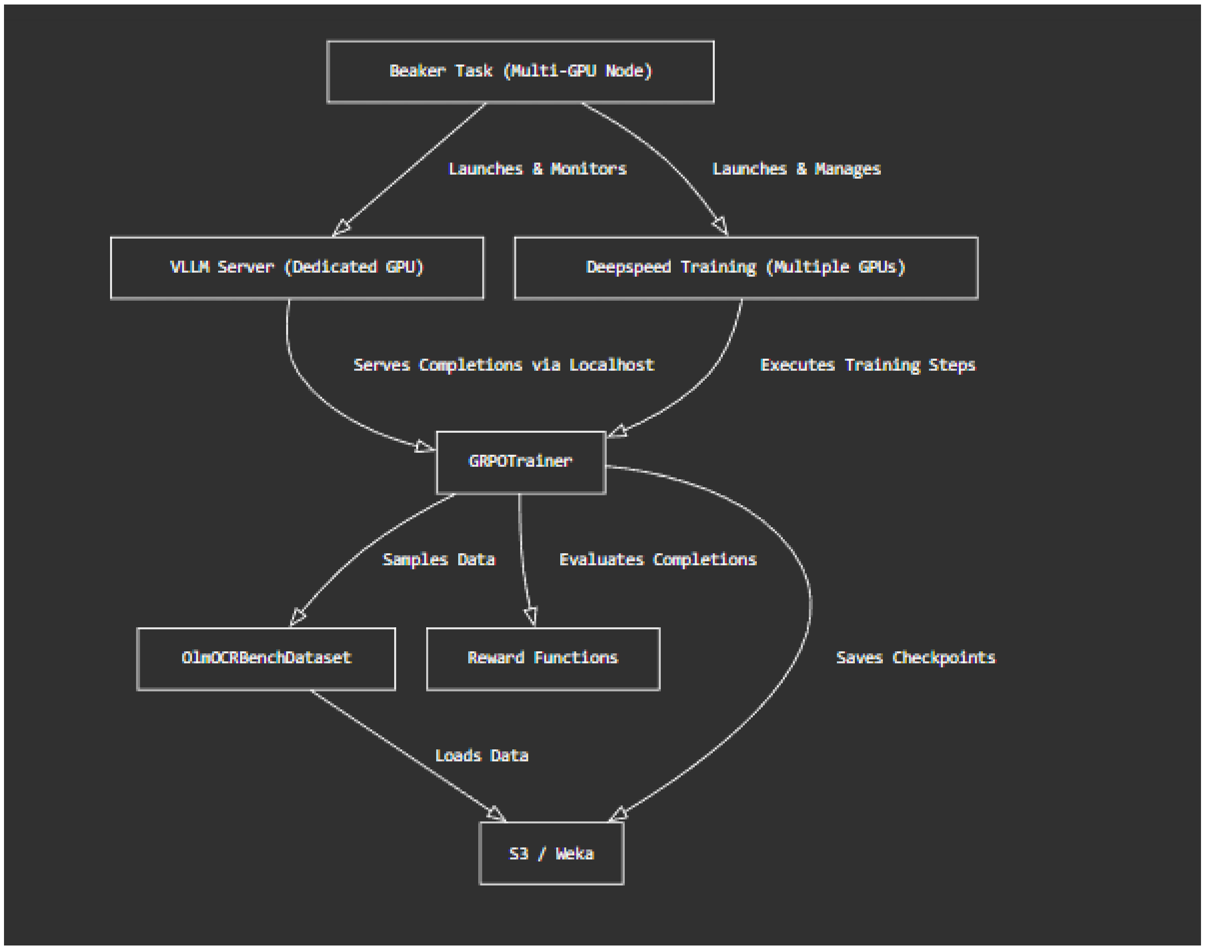
For optimizing model performance, the framework supports a custom optimizer called **SingleDeviceMuonWithAuxAdam**, implemented in **olmocr/train/muon.py**. This optimizer combines a Muon optimizer for specific parameter groups (e.g., hidden weight layers) with the Adam optimizer for others (e.g., embeddings, biases), enabling a nuanced approach to parameter updates, especially in distributed training environments. The core training loop, managed by **olmocr/train/train.py**, integrates these components, handling aspects like gradient accumulation, mixed-precision training, learning rate scheduling, and comprehensive logging to platforms like **wandb**.

Following initial training, checkpoints undergo preparation and quantization to ready them for deployment. The **olmocr/train/prepare_checkpoint.py** script facilitates merging LoRA (Low-Rank Adaptation) adapters into base models, averaging weights from multiple checkpoints (model souping), and managing the transfer of model files between local storage and S3. For further optimization, **olmocr/train/compress_checkpoint.py** performs FP8 dynamic quantization using **llmcompressor**, which significantly reduces model size and inference time with minimal impact on accuracy. This process can optionally use a calibration dataset derived from PDFs to inform the quantization process.

Validation of these prepared models is crucial. The script **olmocr/train/compare_vllm_checkpoint.py** meticulously compares the token-level outputs and probabilities of models running on **vLLM** (a high-performance inference engine) against their HuggingFace counterparts. This comparison helps identify any discrepancies in behavior, ensuring consistency and reliability across different deployment environments.

Orchestration of these training and validation experiments, particularly for GRPO models, is managed via Beaker, a platform for machine learning experiments. Shell scripts within **scripts/train** automate the submission and execution of multi-GPU training jobs. These scripts handle Docker image management, environment setup, and data synchronization from S3, ensuring that experiments are reproducible and scalable. A key design choice involves co-locating a **vLLM** server and the training process within a single Beaker task, optimizing resource utilization and inter-process communication for multi-GPU setups. These orchestrators dynamically generate Python scripts to interact with the Beaker API, configure **ExperimentSpec** and **TaskSpec**, and manage resource allocation, including specialized configurations for InfiniBand networking for high-performance clusters. Further details on managing experiment environments can be found in Beaker Experiment Orchestration and Environment Setup.

Specialized GRPO Training and Multi-GPU Orchestration



GRPO (Group Relative Policy Optimization) training in olmocr focuses on fine-tuning multimodal models for PDF document understanding, particularly in environments like Beaker. This advanced training leverages multiple reward functions and orchestrates multi-GPU processing, integrating a VLLM (Visual Language Model) server and DeepSpeed within a single Beaker task.

The training process is orchestrated by shell scripts such as scripts/train/grpotrainer-beaker-multi-gpu-augusta.sh and scripts/train/grpotrainer-beaker-multi-gpu.sh, which automate the submission and execution of GRPO experiments on the Beaker platform. These scripts manage Docker image creation, environment setup, and data synchronization from S3, ensuring a consistent and reproducible training environment.

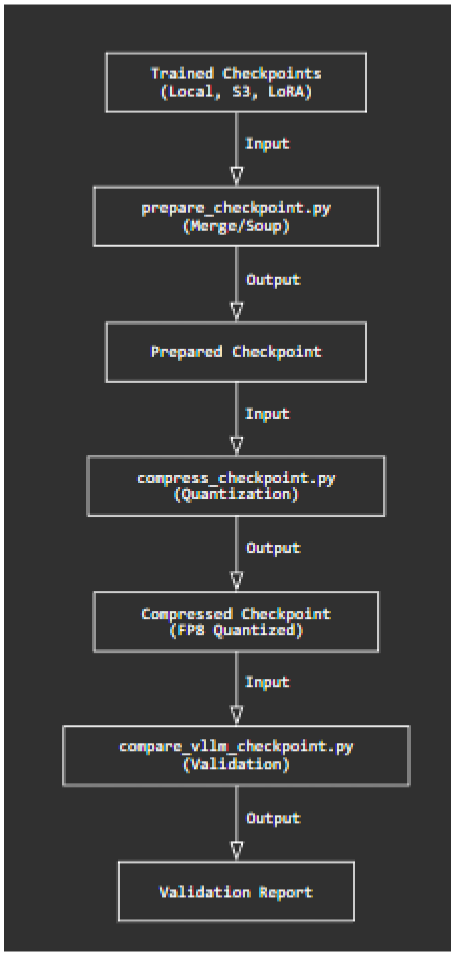
A key aspect of this multi-GPU orchestration is the co-location of a trl vllm-serve instance and the distributed training processes within a single Beaker task. One GPU is dedicated to running the VLLM server in the background, which provides fast inference, while the remaining GPUs are utilized for DeepSpeed-enabled distributed training, launched via **accelerate launch**. This setup ensures that the VLLM server is operational before training begins, facilitating efficient communication between the components via **localhost**.

The GRPO training itself, implemented in **olmocr/train/grpo_train.py**, leverages the **GRPOTrainer** from the **trl** library. It incorporates various reward functions, such as **olmocr_bench_reward**, **medoid_reward**, **bench_edit_distance_reward**, **reward_front_matter**, **reward_element_count**, and **reward_eos**. These functions provide comprehensive feedback by evaluating model completions against document-understanding benchmarks, edit distance to reference outputs, front matter parsing accuracy, and structural element counts. This modular approach to reward functions allows for flexible experimentation and optimization of the model's behavior. The **OlmOCRBenchDataset** class handles the loading of PDF pages and associated metadata, including rendering PDFs to images and formatting prompts for the multimodal model.

Environment setup is crucial for these complex training tasks. The Beaker scripts handle installing necessary Python dependencies, synchronizing datasets from S3, and configuring environment variables such as **WANDB_API_KEY** and AWS credentials. This ensures that the training environment is properly configured for distributed operations and integrates with tools like Weights & Biases for experiment tracking. A detailed guide for setting up the environment and preparing datasets is also available in **olmocr/train/README.md**.

For a broader understanding of how model training and validation are generally handled, refer to Model Training and Evaluation. The underlying configuration mechanisms, including pipeline steps and dataset parameters, are defined in **olmocr/train/config.py**. The training loop, LoRA integration, and checkpointing strategies are implemented in **olmocr/train/train.py**. The optimization process, including the custom **SingleDeviceMuonWithAuxAdam** optimizer, is detailed in **olmocr/train/muon.py**.

Checkpoint Preparation, Quantization, and Validation



The olmOCR project employs a comprehensive workflow for preparing trained models for deployment, which involves merging LoRA (Low-Rank Adaptation) adapters, performing model souping, and applying FP8 dynamic quantization. This process ensures that models are optimized for inference, balancing performance with efficiency, and are validated against existing HuggingFace models.

Checkpoint preparation is managed by the olmocr/train/prepare_checkpoint.py script. This script handles various scenarios for model deployment. If a checkpoint includes LoRA adapters, the script merges these adapters into the base model, converting the parameter-efficient fine-tuned model into a standalone, deployable model. It detects the model architecture by analyzing the config.json file and supports architectures like Qwen2VLForConditionalGeneration and Qwen2_5_VLForConditionalGeneration.

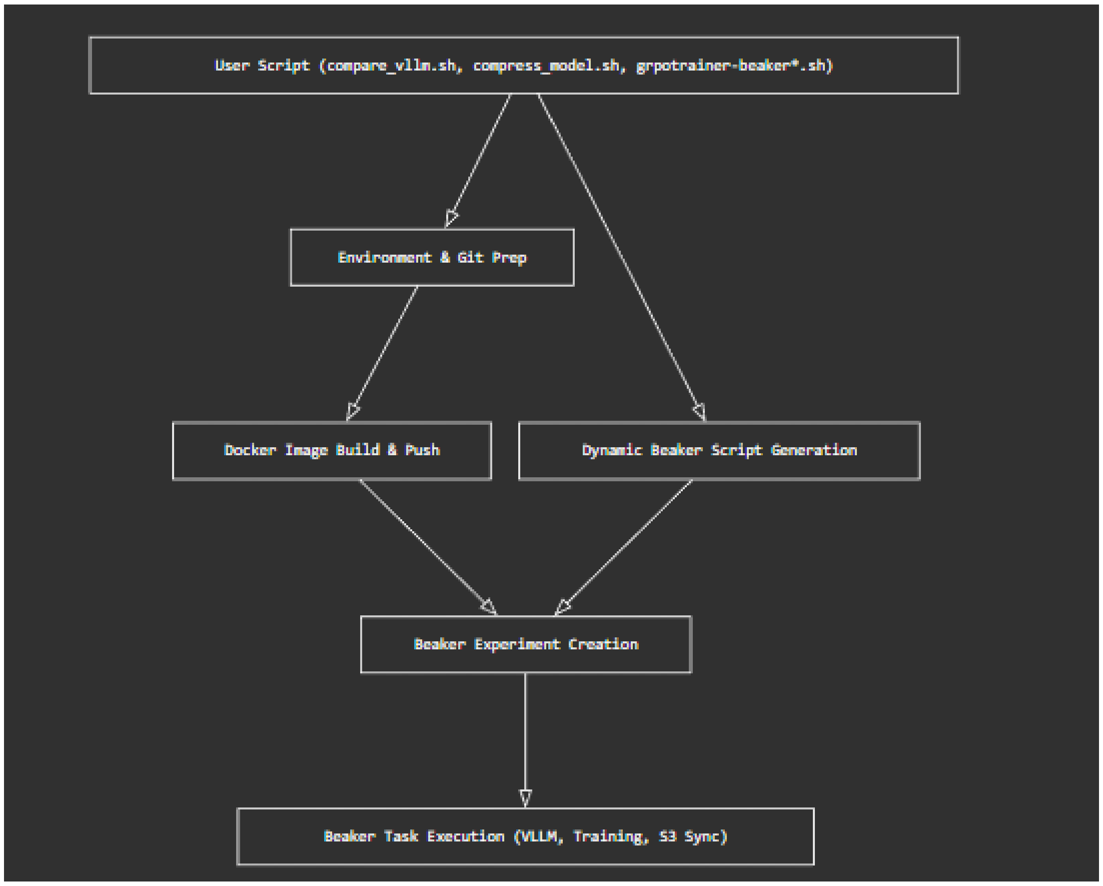
The merging process leverages the **peft** library, and the script also includes mechanisms to clean up GPU memory after the merge.

For scenarios involving multiple trained checkpoints, the system supports "model souping," where the weights of several models are averaged to create a single, more robust model. This process involves downloading all source checkpoints, identifying weight files (e.g., **.safetensors**, **.bin**), and then calculating the element-wise average of their parameters. The averaged weights are saved, and the script handles data transfer between local storage and S3 using **boto3** and **smart_open**. Additionally, it downloads necessary tokenizer files from Hugging Face based on the detected model architecture to ensure the deployed model is self-contained.

After a model is prepared, it can undergo FP8 dynamic quantization using the **olmocr/train/compress_checkpoint.py** script. This process reduces the precision of model weights and activations to FP8 format, which can significantly reduce memory footprint and improve inference speed with minimal impact on performance. The quantization can optionally use a calibration dataset, which is prepared by selecting a subset of PDF documents and extracting multimodal inputs (images and text) to inform the quantization process. The script supports loading and saving models from both local paths and S3, utilizing temporary directories for staging S3 operations.

To ensure the integrity and performance of the quantized models, validation is performed by comparing **vLLM** inference outputs against HuggingFace models using the **olmocr/train/compare_vllm_checkpoint.py** script. This script loads image-text prompts from a benchmark dataset and compares the token-level outputs and probabilities generated by both **vLLM** and HuggingFace implementations of the model. It identifies discrepancies, such as token mismatches or significant probability differences, and provides aggregated statistics on the comparison, helping to verify that quantization or other optimizations do not adversely affect model behavior. This comparison process can be orchestrated via a Beaker experiment, as detailed in Beaker Experiment Orchestration and Environment Setup, using the **scripts/compare_vllm.sh** and **scripts/compress_model.sh** scripts.

Beaker Experiment Orchestration and Environment Setup


User Script (compare_vllm.sh, compress_model.sh, grpotrainer-beaker*.sh)
Environment & Git Prep
Docker Image Build & Push
Dynamic Beaker Script Generation
Beaker Experiment Creation
Beaker Task Execution (VLLM, Training, S3 Sync)

The olmOCR project orchestrates the submission and execution of training experiments, including specialized GRPO (Generative Retrieval with Policy Optimization) and OlmoOCR models, on the Beaker platform. This automation extends to managing Docker images, configuring the execution environment, orchestrating multi-GPU setups (including vLLM servers and DeepSpeed), and synchronizing data from S3.

The process begins by preparing the environment, which involves parsing command-line arguments to distinguish between Beaker-specific parameters and arguments for the underlying Python training script. Crucially, the system retrieves version information and Git hashes to ensure consistent Docker image tagging and to maintain reproducibility of experiments. A Docker image, which encapsulates all necessary dependencies and code, is built and pushed to Beaker.

A key design element is the dynamic generation of a temporary Python script that interacts with the Beaker SDK. This script defines the **ExperimentSpec** and **TaskSpec**, allowing for programmatic and flexible configuration of complex Beaker experiments. Within this configuration, the script specifies the Docker image to be used, the **bash** commands to be executed within the Beaker task (including dependency installation, data synchronization from S3 using **s5cmd**, and the execution of the training command), and the required computational resources (e.g., GPU count, shared memory). It also sets constraints for target clusters and configures environment variables, including sensitive information passed as Beaker secrets. Data volumes, such as WEKA filesystems, are mounted to provide access to large datasets.

For multi-GPU GRPO training, the orchestration specifically handles the setup of a **vLLM** server on one GPU while the remaining GPUs are allocated for **accelerate**-launched DeepSpeed training. This co-location within a single Beaker task simplifies resource management and inter-process communication. Robustness is ensured through mechanisms like **cleanup** functions that sync outputs to S3 even if training fails, and wait loops that ensure the **vLLM** server is ready before training commences.

Beyond Beaker, the framework also includes scripts for training on HPC clusters like Frontier, utilizing SLURM for resource allocation, loading necessary environment modules, and configuring Hugging Face for offline operation and shared caches.

The orchestration of these experiments on Beaker is achieved through several scripts, such as **scripts/train/grpotrainer-beaker-multi-gpu-augusta.sh**, **scripts/train/grpotrainer-beaker-multi-gpu.sh**, **scripts/train/grpotrainer-beaker.sh**, and **scripts/train/newtrainer-beaker.sh**, which automate the submission of training jobs. Similarly, the comparison of VLM inference between **vLLM** and HuggingFace checkpoints, as well as model compression workflows, are also orchestrated via Beaker using scripts like **scripts/compare_vllm.sh** and **scripts/compress_model.sh** respectively. These scripts dynamically generate Python code to interact with the Beaker API, streamlining the entire experiment lifecycle from image management to resource allocation and execution. Further details on data management and preparation can be found in Data Management and Processing.