Towards AI

Member-only story

Deploy an in-house Vision Language Model to parse millions of documents: say goodbye to Gemini and OpenAI.

How to build a Document Parsing Pipeline to process millions of documents using Qwen-2.5-VL, vLLM, and AWS Batch.

Jeremy Arancio Follow 18 min read · Apr 23, 2025

1.3K 17

*deployed as a **Batch Inference** pipeline on **AWS Batch** with **Terraform**. Surprisingly, our in-house approach was **cheaper than third-party LLM providers**, even without basic optimizations.*

Nowadays, most companies develop their AI features using external LLM providers.

OpenAI with GPT, Google with Gemini, Anthropic with Claude. One API call. That's it, your application is **AI featured!**

However, this ease of development comes with a price that most companies, that already passed the POC stage, are not ready to pay…

- **Data security:** your data is their data. Used to train the next LLM version, or to perform obscure analysis on your usage.

- **Costs: w**hen you use an LLMs from an API endpoint, you pay per token. This means the cost grows linearly: the more calls, the more expensive. While the price seems low for simple POCs, it can rapidly grow out of control.
- **Reliability:** you have no control over LLM uptimes and downtimes. The server fails due to high demand? That's your all feature (or application) that shuts down. Good luck for sending a support ticket.
- **Performances: P**rediction accuracies depends on the size of the model and the quality of your prompt. But let's be real: you can prompt/beg the LLM as much as you want, it just haven't seen enough of your data during its training to be performant enough on your task…
- **Maintainability:** One day, everything works fine. Your prompts are set up. The next day, nothing works anymore. You need to recalibrate everything. Again. Until the next day.

> *However, with the recent development in the open-source community, it became possible for anyone to build a complete in-house LLM feature and solve all these problems.*

To showcase how to do it, we take a common problem that I often encountered in my Machine Learning Engineer career: **Extracting data from documents.**

In this article, you'll learn how to deploy an open-source Vision LLM, **Qwen-2.5-VL** on **AWS Batch.**

Buckle up! You'll never consider OpenAI again for developing your AI features after reading this article.

GitHub - jeremyarancio/VLM-Batch-Deployment: Batch Deployment for Document Parsing with AWS Batch &...

Batch Deployment for Document Parsing with AWS Batch & Qwen-2.5-VL - jeremyarancio/VLM-Batch-Deployment

github.com

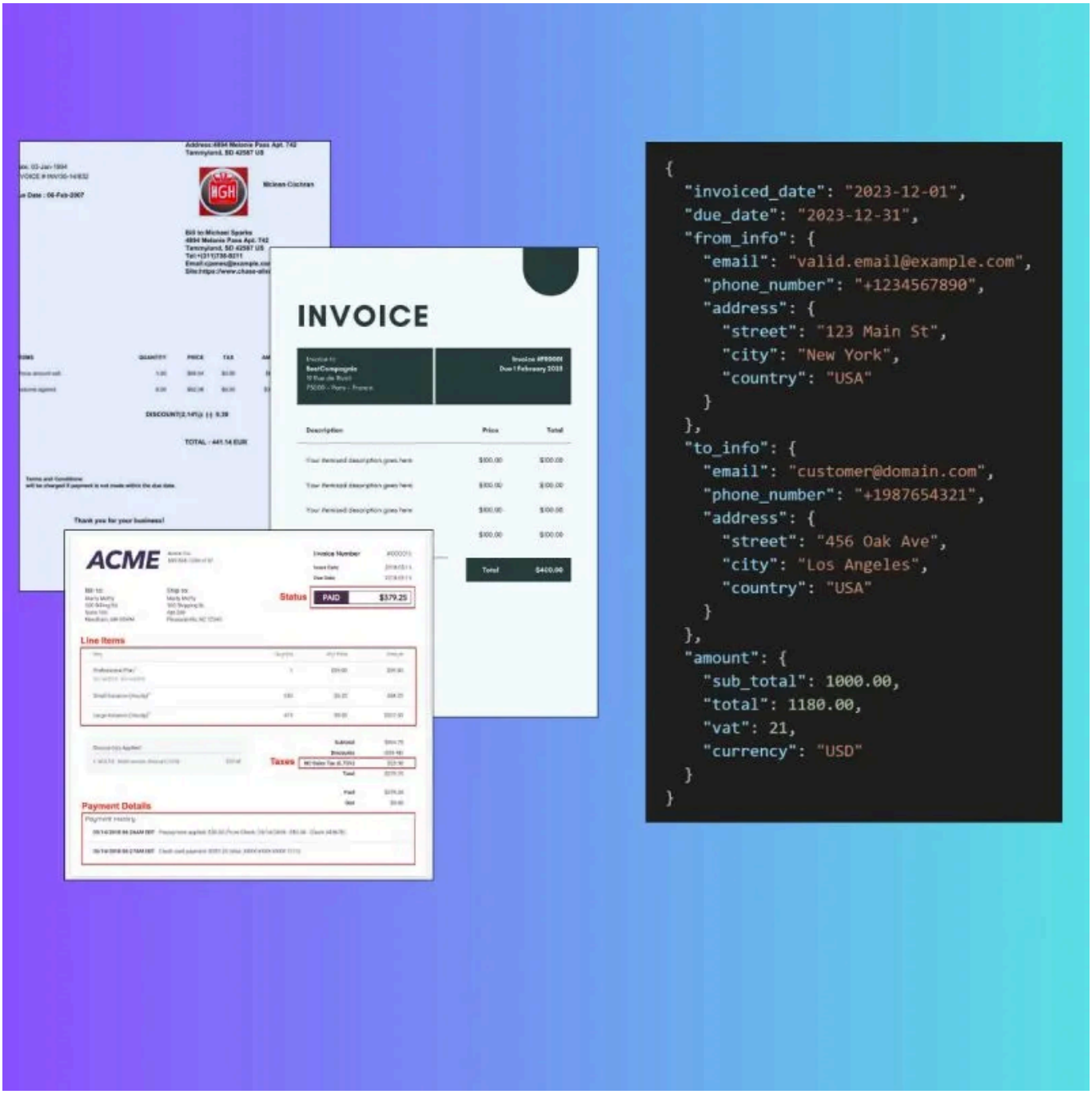
The feature components

Parsing documents: a job for Vision Language Models!

A Vision Language Model (VLM) is a category of Transformer's architecture trained on text **and** images. Instructions and images are embedding separately, then joined during the model forward path.

This opens many possibilities, such as image and video recognition coupled with user instructions.

Our task is to extract structured document data from the model's generation.



We have several candidates for this task:

- Qwen-2.5-VL

A model developed by Alibaba, one of the most performant amongst open-source models. The VL version proposes many features, such as image

understanding, long-video processing, object recognition, and structured outputs.

The collections of models, from 3 to 72 billions parameters, is available on the Hugging Face platform.

- **SmolVLM**

An open-source model developed by Hugging Face. It represents the lightest model out of all VLMs, with only 256 millions parameters! The architecture is composed of SigLip for the encoder part and SmolLM2 for the decoder part.

The model was trained on image understanding, algebraic reasoning, table understanding and more… Find more about SmolVLM on its official Hugging Face model page.

- **Idefics3**

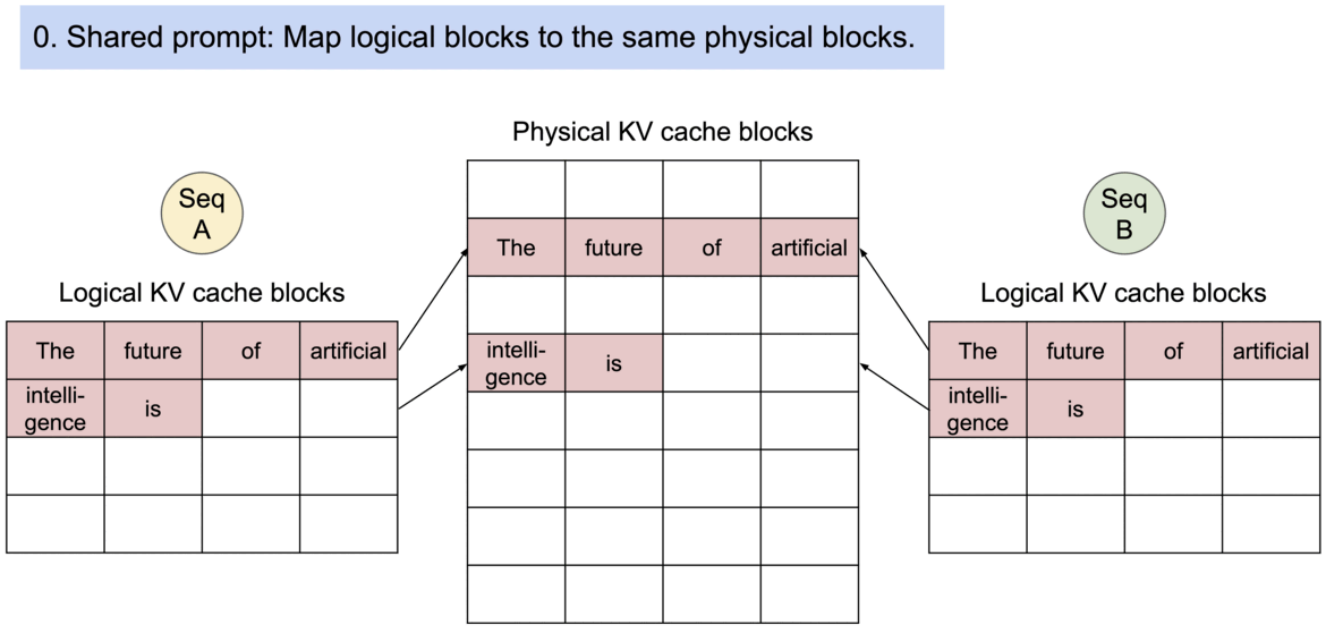
Another model from Hugging Face.

Idefics3 is a fusion between SigLip and Llama-3.1. It was trained on open datasets about image understand and Question-Answer instructions.

Since **Qwen-2.5-VL** is already fine-tuned to return structured outputs, it makes it an ideal candidate for parsing documents.

vLLM to serve the VLM

vLLM is an LLM serving tool that drastically accelerates LLM inferences. It leverages **PagedAttention,** a mechanism that allocates optimally the KV cache in memory to process several LLM requests at the same time.



PagedAttention (Source)

vLLM not only supports text-only models, it also supports Vision Language Models, in addition to structured outputs generation, which makes it a the perfect fit.

The best part of it: it is directly integrated with the Transformers library. This allows us to load any model from the Hugging Face platform and vLLM automatically assign the model weights into the GPU VRAM.

There are 2 deployment configurations we can use with vLLM: **online inference** and **offline inference.**

In this article, we'll only focus on the offline inference. We'll create a Python module that uses vLLM and process images into structured outputs like

JSON.

AWS Batch to run the inference

The feature we'll develop will be used as a step in an ETL (Extract — Transform — Load) pipeline. On a regular basis, let's say once a day, the model will extract data from thousands of images, until its completion.

This is the perfect use case for Batch Deployment.

We'll use AWS Batch to manage batch jobs in the cloud.

It is composed as follows:

🏗️ **Jobs**

The smallest entities. Based on Docker containers, a Job runs a module until its completion or failure.

Compared to serving, which runs computations 24/7, it's a cost-effective solution for time-bounded tasks.

🏗️ **Job Definition**

The job template. It defines the resource allocated for the task, the location of the Docker Image, the number of retries, priorities, and maximum time.

Once defined, you are able to create and run as many jobs as you like.

🏗️ **Job queue**

Responsible for orchestrating and prioritizing jobs based on resources availability and job priorities.

You can create a multitude of queues for high-priority tasks, for time-sensitive jobs for example, or low-priority, which can run on cheapest resources.

Resources are allocated to each queue via the Computation Environment.

🏗️ Computation Environment

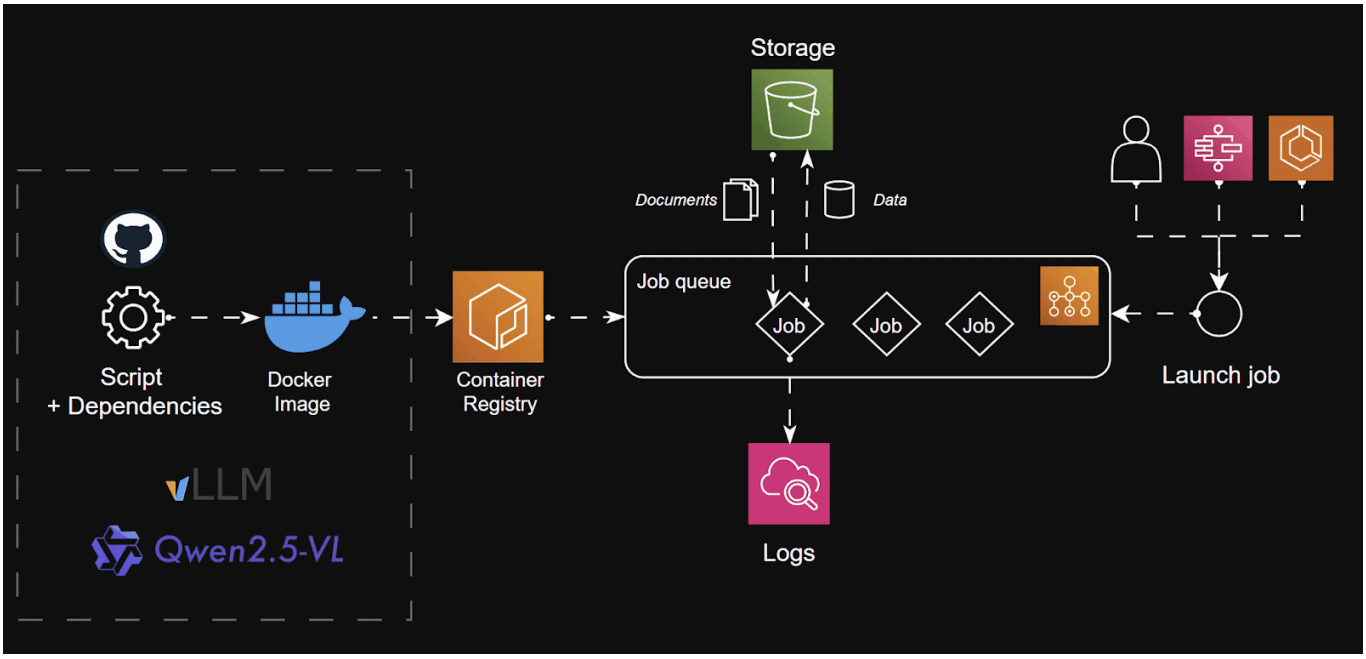
It defines the resources you're ready to allocate for all your job queues. GPUs for heavy computation tasks such as LLM training, or Spot Instance for fault-tolerant tasks.

Job queue automatically allocate the right amount of resources based on the Job Definition requirements.

AWS Batch is a powerful service from AWS to run any type of job, using any resources from EC2: CPU and GPU instances. You only pay for the running time. Once the job is finished, AWS Batch automatically turns off everything. This makes it the perfect candidate for our use case.

Let's start building our Document Parsing feature

The VLM deployment will look like this:



- Because of its capacity of generating structured output straight out of the box, we'll use **Qwen-2.5-VL** to perform the data extraction from documents.
- To accelerate the inference, we use **vLLM** to automatically manage large batch inferences. Indeed, vLLM automatically handles large numbers of tokens coming from instructions and embedded images using the **PagedAttention** mechanism.
- The scripts and modules containing vLLM offline inference will be packaged using **uv** and containerized using Docker. This makes the module deployable almost anywhere: AWS Batch / GCP Batch / Kubernetes.
- The Docker Image will be stored on **AWS ECR** to keep the feature in the AWS ecosystem.
- We use **AWS Batch** to manage the job queues. We'll orchestrate the instance using EC2 instead of Fargate. The reason being Fargate doesn't propose GPU resources, which are essential for running LLMs. The infrastructure will be managed and deployed using **Terraform.**

- We'll use **AWS S3** as our storage solution to download documents and upload the extracted data as a dataset. The data can then be used in any ETL pipeline such as feeding analytics dashboards.

Let's start!

The job module

We begin with writing the job script.

The process goes as follows:

1. Documents, stored in an S3 bucket as images, are downloaded using the AWS SDK. We also identify each document by its S3 path, which is unique. This will help us identify the data output with its respective document.

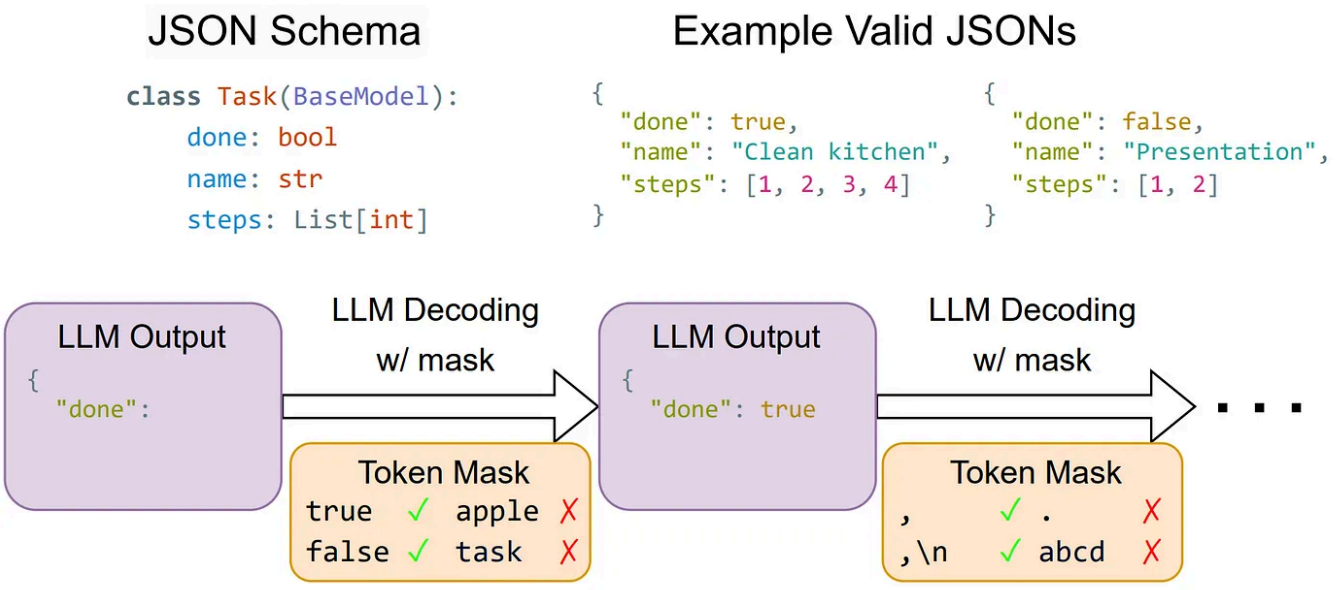
```
#llm/parser/main.py
from io import BytesIO
from PIL import Image
import logging

import boto3
from botocore.exceptions import ClientError

LOGGER = logging.getLogger(__name__)

def load_images(
    s3_bucket: str,
    s3_images_folder_uri: str,
) -> tuple[list[str], list[Image.Image]]:
    try:
        s3 = boto3.client("s3")
        response = s3.list_objects_v2(Bucket=s3_bucket, Prefix=s3_images_folder_

        images: list[Image.Image] = []
        filenames: list[str] = []
```

```
        for obj in response["Contents"]:
            key = obj["Key"]
            filenames.append(key)
            response = s3.get_object(Bucket=s3_bucket, Key=key)
            image_data = response["Body"].read()
            images.append(Image.open(BytesIO(image_data)))
        return filenames, images
    except ClientError as e:
        LOGGER.error("Issue when loading images from s3: %s.", str(e))
        raise
    except Exception as e:
        LOGGER.error("Something went wrong when loading the images from AWS S3:
        raise
```

2. We use vLLM to load the model from the Hugging Face platform and prepare it for the GPU.

It also comes with useful features, such as **Guided decoding**, which can be set in the `SamplingParams` parameter.

In the back-end, vLLM uses the **xGrammar** mechanism to orient the text generation output into a valid JSON format.



Constrained decoding by xgrammar (source)

By providing a Pydantic schema of the expected output, we can guide the generation into the right data format.

```python
#llm/parser/schemas.py
import re

from pydantic import BaseModel, field_validator, ValidationInfo


class Address(BaseModel):
    street: str | None = None
    city: str | None = None
    country: str | None = None


class Info(BaseModel):
    email: str | None = None
    phone_number: str | None = None
    address: Address


class Amount(BaseModel):
    sub_total: float | None = None
    total: float | None = None
    vat: float | None = None
    currency: str | None = None


class Invoice(BaseModel):
    invoiced_date: str | None = None
    due_date: str | None = None
    from_info: Info
    to_info: Info
    amount: Amount
```

```python
#llm/parser/main.py
from vllm import LLM, SamplingParams
from vllm.sampling_params import GuidedDecodingParams
from pydantic import BaseModel
```

```
from llm.settings import settings


def load_model(
    model_name: str, schema: Type[BaseModel] | None
) -> tuple[LLM, SamplingParams]:
    llm = LLM(
        model=model_name,
        gpu_memory_utilization=settings.gpu_memory_utilisation,
        max_num_seqs=settings.max_num_seqs,
        max_model_len=settings.max_model_len,
        mm_processor_kwargs={"min_pixels": 28 * 28, "max_pixels": 1280 * 28 * 28
        disable_mm_preprocessor_cache=True,
    )
    sampling_params = SamplingParams(
        guided_decoding=GuidedDecodingParams(json=schema.model_json_schema())
        if schema
        else None,
        max_tokens=settings.max_tokens,
        temperature=settings.temperature,
    )
    return llm, sampling_params
```

3. Once the model and images are loaded in memory, we can run the inference using vLLM.

Since we use Guided Decoding instead of vanilla inference, the process will take a bit more time in exchange for better data quality.

To assist the text generation, we also provide an instruction guiding the LLM to return the valid schema. **We need to fit the instruction into the prompt template used during Qwen-2.5-VL fine-tuning.**

```
#llm/parser/prompts.py
INSTRUCTION = """
Extract the data from this invoice.
Return your response as a valid JSON object.
```

```
Here's an example of the expected JSON output:

{
    "invoiced_date": 09/04/2025  # format DD/MM/YYYY
    "due_date": 09/04/2025  # format DD/MM/YYYY
    "from_info": {
        "email": "jeremya@gmail.com",
        "phone_number": "+33645789564",
        "address": {
            "street": "Chemin des boulangers",
            "city": "Bourges",
            "country": FR # 2 letters country
        },
    "to_info": {
        "email": "igordosgor@gmail.com",
        "phone_number": "+33645789564",
        "address": {
            "street": "Chemin des boulangers",
            "city": "New York",
            "country": US
        },
    }
    "amount": {
        "sub_total": 1450.4 # Before taxes
        "total": 1740.48 # After taxes
        "vat": 0.2 # Pourcentage
        "currency": USD # 3 letters code (USD, EUR, ...)
    }
}
""".strip()


QWEN_25_VL_INSTRUCT_PROMPT = (
    "<|im_start|>system\nYou are a helpful assistant.<|im_end|>\n"
    "<|im_start|>user\n<|vision_start|><|image_pad|><|vision_end|>"
    "{instruction}<|im_end|>\n"
    "<|im_start|>assistant\n"
)
```

```
#llm/parser/main.py
from PIL import Image
from vllm import LLM, SamplingParams

from llm.parser import prompts
```

```
def run_inference(
    model: LLM, sampling_params: SamplingParams, images: list[Image.Image], prom
) -> list[str]:
    """Generate text output for each image"""
    inputs = [
        {
            "prompt": prompt,
            "multi_modal_data": {"image": image},
        }
        for image in images
    ]
    outputs = model.generate(inputs, sampling_params=sampling_params)
    return [output.outputs[0].text for output in outputs]


if __name__ == "__main__":
    outputs = run_inference(
        model=model,
        sampling_params=sampling_params,
        images=images,
        prompt=prompts.QWEN_25_VL_INSTRUCT_PROMPT.format(
            instruction=prompts.INSTRUCTION
        ),
    )
```

4. Once the inference over, we extract and validate the JSON from the generated text. Qwen-2.5-VL coupled with xgrammar already done a great job to avoid any excedent texts!

To do so, we simply use the `json` Python library to decode the JSON:

```
#llm/parser/main.py
import json
import logging
from typing import Any

LOGGER = logging.getLogger(__name__)

def extract_structured_outputs(outputs: list[str]) -> list[dict[str, Any]]:
    json_outputs: list[dict[str, Any]] = []
    for output in outputs:
```

```
            start = output.find("{")
            end = output.rfind("}") + 1  # +1 to include the closing brace
            json_str = output[start:end]
            try:
                json_outputs.append(json.loads(json_str))
            except json.JSONDecodeError as e:
                LOGGER.error("Issue with decoding json for LLM output: %s", e)
                json_outputs.append({})
        return json_outputs
```

We also validate the returned JSON with Pydantic.

Here, Pydantic becomes super handy since it enables the module to return a valid schema despite the generation. LLMs tend to “hallucinate”, which could be dramatic in a production application.

However, Pydantic doesn’t come with an out-of-the-box “return default_value if invalidate”.

For this, we’ll create our own validation using `@field_validator` decorator. Check the documentation to know more about the feature.

```
class Info(BaseModel):
    email: str | None = None
    phone_number: str | None = None
    address: Address

    @field_validator("email", mode="before")
    @classmethod
    def validate_email(cls, email: str | None, info: ValidationInfo) -> str | No
        if not email:
            return None
        email_pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
        if not re.match(email_pattern, email):
            return cls.model_fields[info.field_name].get_default()
```

```
        else:
            return email
```

5. Finally, the data is converted into a dataset, with the unique identifier for each element (we picked the S3 path of each document).

We use **uv** to manage the Python dependencies and package our application. Additionally, using the Pyproject.toml, we add the cli command `run-batch-job` to run the job script.

```
#pyproject.toml
[project.scripts]
run-batch-job = "llm.__main__:main"
```

Once the package is built, we just have to run:

```
uv run run-batch-job
```

Regarding the settings of the application, such as the S3 bucket name and the S3 prefixes, we use `pydantic-settings` , instead of a basic Python class. The reason being Pydantic automatically validates the job configuration, in addition to validating the environment variables.

```
from typing import Annotated

from pydantic_settings import BaseSettings, SettingsConfigDict
```

```
from pydantic import Field


class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file=".env", extra="ignore"
    )  # extra="ignore" for AWS credentials

    # Model config
    model_name: str = "Qwen/Qwen2.5-VL-3B-Instruct"
    gpu_memory_utilisation: Annotated[float, Field(gt=0, le=1)] = 0.9
    max_num_seqs: Annotated[int, Field(gt=0)] = 2
    max_model_len: Annotated[int, Field(multiple_of=8)] = 4096
    max_tokens: Annotated[int, Field(multiple_of=8)] = 2048
    temperature: Annotated[float, Field(ge=0, le=1)] = 0

    # AWS S3
    s3_bucket: str
    s3_preprocessed_images_dir_prefix: str
    s3_processed_dataset_prefix: str


settings = Settings()
```

S3 settings are implemented during runtime, meaning we can modify the settings without re-building the package!

Containerize the module with Docker

We use the official multi-stage build from the **uv** documentation..

(The Docker Image size is ~9GB because of the Pytorch and Cuda libraries!)

```
# An example using multi-stage image builds to create a final image without uv.

# First, build the application in the `/app` directory.
# See `Dockerfile` for details.
FROM ghcr.io/astral-sh/uv:python3.12-bookworm-slim AS builder
ENV UV_COMPILE_BYTECODE=1 UV_LINK_MODE=copy

# Disable Python downloads, because we want to use the system interpreter
```

```
# across both images. If using a managed Python version, it needs to be
# copied from the build image into the final image; see `standalone.Dockerfile`
# for an example.
ENV UV_PYTHON_DOWNLOADS=0

WORKDIR /app
RUN --mount=type=cache,target=/root/.cache/uv \
    --mount=type=bind,source=uv.lock,target=uv.lock \
    --mount=type=bind,source=pyproject.toml,target=pyproject.toml \
    uv sync --frozen --no-install-project --no-dev
ADD . /app
RUN --mount=type=cache,target=/root/.cache/uv \
    uv sync --frozen --no-dev


# Then, use a final image without uv
FROM python:3.12-slim-bookworm

# Copy the application from the builder
COPY --from=builder --chown=app:app /app /app

# vLLM requires some basic compilation tools
RUN apt-get update && apt-get install -y build-essential

# Place executables in the environment at the front of the path
ENV PATH="/app/.venv/bin:$PATH"

# Executable package. Check pyproject.toml
CMD ["run-batch-job"]
```

Once built, we upload the Docker Image to ECR. We implement the logic directly into a **Makefile** to make the command easy to run. This also helps document the project.

```
AWS_REGION ?= eu-central-1
ECR_REPO_NAME ?= demo-invoice-structured-outputs
ECR_URI = ${ECR_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/${ECR_REPO_NAME}

.PHONY: deploy
deploy: ecr-login build tag push

 # Require ECR_ACCOUNT_ID (fail if not provided)
```

```
ifndef ECR_ACCOUNT_ID
$(error ECR_ACCOUNT_ID is not set. Please provide it, e.g., `make deploy ECR_AC
endif

ecr-login:
@echo "[INFO] Logging in to ECR..."
aws ecr get-login-password --region ${AWS_REGION} | docker login --username AWS
@echo "[INFO] ECR login successful"

build:
@echo "[INFO] Building Docker image..."
docker build -t ${ECR_REPO_NAME} .
@echo "[INFO] Build completed"

tag:
@echo "[INFO] Tagging image for ECR..."
docker tag ${ECR_REPO_NAME}:latest ${ECR_URI}:latest
@echo "[INFO] Tagging completed"

push:
@echo "[INFO] Pushing image to ECR..."
docker push ${ECR_URI}:latest
@echo "[INFO] Push completed"
```

You can then run the next command.

```
make deploy ECR_ACCOUNT_ID=<YOUR_ECR_ACCOUNT_ID>
```

Deploy AWS Batch with EC2 orchestration

AWS Batch proposes 3 main approaches to perform jobs using AWS resources: EC2, Fargate, or EKS. Since Fargate doesn't allow the GPU usage and EKS requires an existing cluster running, we opt for the EC2 orchestration.

We'll explain the deployment using **Terraform**, an Infrastructure As a Code tool.

IAM Roles

First comes **IAM** permissions and roles.

In AWS, **roles enable an** AWS service to interact with other. AWS Batch requires **4** roles:

- **EC2 instance role**

AWS Batch actually uses ECS under the hood to launch and run EC2 instances. Therefore, we need to create a policy to access the EC2 service and assign this role to an ECS role, that will be assign to **AWS Batch Compute Environnment**

> *I know, why make it simple if we can make it complex*

```
# IAM Policy Document
data "aws_iam_policy_document" "ec2_assume_role" {
  statement {
    effect = "Allow"

    principals {
      type        = "Service"
      identifiers = ["ec2.amazonaws.com"]
    }

    actions = ["sts:AssumeRole"]
  }
}

# IAM Role Creation
resource "aws_iam_role" "ecs_instance_role" {
  name               = "ecs_instance_role"
  assume_role_policy = data.aws_iam_policy_document.ec2_assume_role.json
```

```
}

# Policy Attachment
resource "aws_iam_role_policy_attachment" "ecs_instance_role" {
  role       = aws_iam_role.ecs_instance_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonEC2ContainerServicefo
}

# Instance Profile Creation. Used in Batch-Compute-Env
resource "aws_iam_instance_profile" "ecs_instance_role" {
  name = "ecs_instance_role"
  role = aws_iam_role.ecs_instance_role.name
}
```

- **AWS Batch service role**

Policy for AWS Batch service role which allows access to related services including EC2, Autoscaling, EC2 Container service, Cloudwatch Logs, ECS and IAM.

It is the backbone of AWS Batch that connects the service to all other AWS services.

```
data "aws_iam_policy_document" "batch_assume_role" {
  statement {
    effect = "Allow"

    principals {
      type        = "Service"
      identifiers = ["batch.amazonaws.com"]
    }

    actions = ["sts:AssumeRole"]
  }
}

resource "aws_iam_role" "batch_service_role" {
  name               = "aws_batch_service_role"
  assume_role_policy = data.aws_iam_policy_document.batch_assume_role.json
```

```
}

resource "aws_iam_role_policy_attachment" "batch_service_role" {
  role       = aws_iam_role.batch_service_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AWSBatchServiceRole"
}
```

- **ECS Task Execution Role**

Since AWS Batch runs within an ECS cluster, it requires the access to ECS Execution Role to access services, such as pulling a container from an ECR private repository.

```
resource "aws_iam_role" "ecs_task_execution_role" {
  name = "ecs_task_execution_role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Effect    = "Allow",
      Principal = { Service = "ecs-tasks.amazonaws.com" },
      Action    = "sts:AssumeRole"
    }]
  })
}

# Attach the standard ECS Task Execution policy
resource "aws_iam_role_policy_attachment" "ecs_task_execution_role" {
  role       = aws_iam_role.ecs_task_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRoleP
}
```

- **Batch Job Role**

Finally, in the case you need the job to call external AWS services, S3 in our case, we need to indicate a role to the running container. This is a better and more secured way to handle AWS credentials instead of using environment variables.

We also attach to this role the ecs-task-execution role as it is managed by ECS as a task.

We finally limit the permission to our bucket only. This to respect the least-prvilege principle.

```
resource "aws_iam_role" "batch_job_role" {
  name = "demo-batch-job-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Effect    = "Allow",
      Principal = { Service = "ecs-tasks.amazonaws.com" },
      Action    = "sts:AssumeRole"
    }]
  })
}

# IAM Policy for S3 Access
resource "aws_iam_policy" "batch_s3_policy" {
  name        = "batch-s3-access"
  description = "Allow Batch jobs to access S3 buckets"

  policy = jsonencode({
    Version = "2012-10-17",
    Statement = [{
      Effect = "Allow",
      Action = [
        "s3:GetObject",
        "s3:PutObject",
        "s3:ListBucket"
      ],
      Resource = [
        "arn:aws:s3:::${var.s3_bucket}",
```

```
          "arn:aws:s3:::${var.s3_bucket}/*"
        ]
      }]
    })
}
```

Compute Environment

AWS Batch uses **Compute Environment** to launch resources from AWS, either from EC2 or Fargate.

It also supports Spot Instances, which are less expensive than on-demand resources but come with the risk of premature termination. For batch jobs, this is an ideal use case, as failed jobs can easily be retried.

```
resource "aws_batch_compute_environment" "batch_compute_env" {
  compute_environment_name = "demo-compute-environment"
  type                     = "MANAGED"
  service_role             = aws_iam_role.batch_service_role.arn # Role associat

  compute_resources {
    type                = "EC2"
    allocation_strategy = "BEST_FIT_PROGRESSIVE"

    instance_type = var.instance_type

    min_vcpus     = var.min_vcpus
    desired_vcpus = var.desired_vcpus
    max_vcpus     = var.max_vcpus

    security_group_ids = [aws_security_group.batch_sg.id]
    subnets            = data.aws_subnets.default.ids
    instance_role      = aws_iam_instance_profile.ecs_instance_role.arn # Role t
  }

  depends_on = [aws_iam_role_policy_attachment.batch_service_role] # To prevent
}
```

Batch Compute Environment automatically assigns jobs to resources within a VPC. In this case, we used our default VPC from AWS.

> *Check the code to see the Terraform part for creating VPC/Subnets/Security Groups.*

We'll use an **EC2 instance g6.xlarge,** containing an Nvidia L4 with **16GB CPU RAM and ~24GB VRAM.**

IMPORTANT INFORMATION:

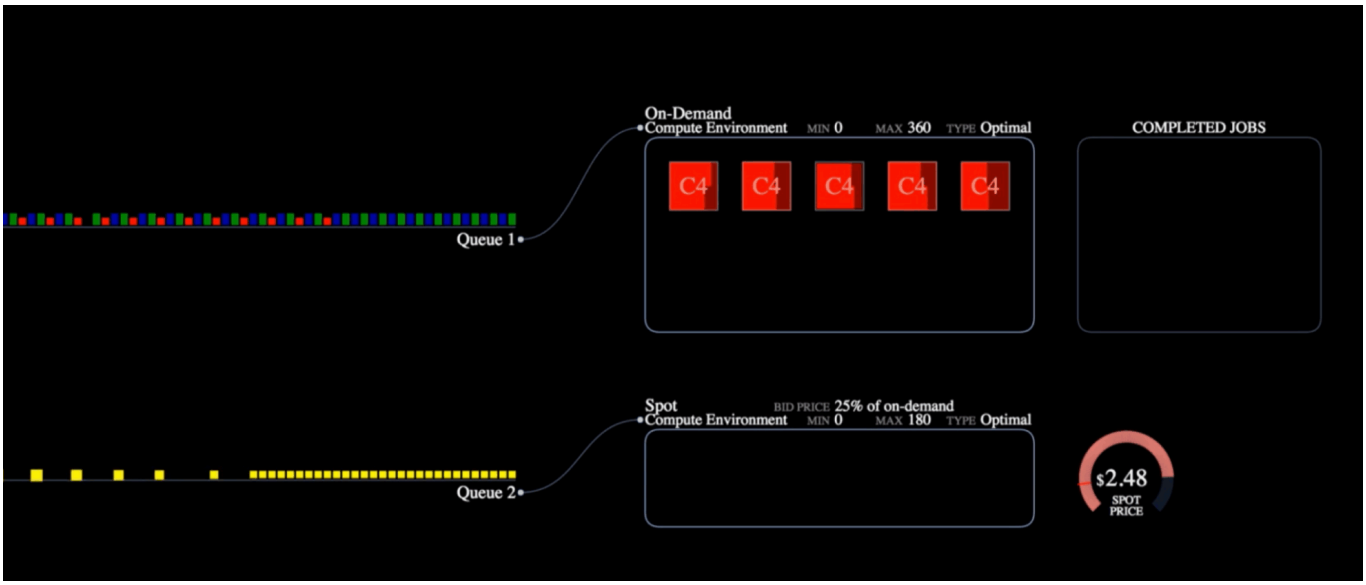
- A g6.xlarge instance is composed of 4 vCPUs, meaning we need to indicate a maximum of 8 vCPUs if we want to run 2 jobs at the same time.
- If the number of *desired_vcpus* is 0, AWS Batch will automatically turn on the required number of instance for a job. *AWS Batch modifies this value between the minimum and maximum values based on job queue demand.*
- Be sure to have the quotas of vCPUs and GPU instance. Check your service quotas and make a request to AWS. *(It caused me some trouble because I didn't have the quotas for g6.xlarge instances. Don't make the same mistake!)*

Job queue

A job queue schedules and manages jobs by priority. When created, we assign a Compute Environment.

```
resource "aws_batch_job_queue" "batch_job_queue" {
  name     = "demo-job-queue"
  state    = "ENABLED"
  priority = 1
  compute_environment_order {
    order               = 1
    compute_environment = aws_batch_compute_environment.batch_compute_env.arn
  }
}
```

You can create several job queues with different priorities and organize your job depending on their own priority.



Job processing in AWS Batch (source)

Job definition

Finally, the job definition. This is the template to create a job. It indicates the priority, where to pull the Docker image from, the environment variables, and more…

It also indicates the resources required by the job. If the resources are available in the compute environment (and not already used in another job queue for example), the job can be launched. Since we run the inference on GPU, we need to indicate the correct number of vCPU, the maximum CPU memory (g6.xlarge has 16GB in total), and number of GPUs (here 1).

```
esource "aws_batch_job_definition" "batch_job_definition" {
  name = "demo-job-definition"
  type = "container"

  container_properties = jsonencode({
    image            = var.docker_image,
    jobRoleArn       = aws_iam_role.batch_job_role.arn,
    executionRoleArn = aws_iam_role.ecs_task_execution_role.arn,
    resourceRequirements = [
      {
        type  = "VCPU"
        value = "4" # g6.xlarge has 4 vCPUs
      },
      {
        type  = "MEMORY"
        value = "8000" # g6.xlarge has 16GB RAM
      },
      {
        type  = "GPU"
        value = "1" # Critical for g6.xlarge
      }
    ],
    environment = [
      {
        name  = "S3_BUCKET",
        value = var.s3_bucket
      },
      {
        name  = "S3_PREPROCESSED_IMAGES_DIR_PREFIX",
        value = var.preprocessed_images_dir_prefix
      },
      {
        name  = "S3_PROCESSED_DATASET_PREFIX",
        value = var.s3_processed_dataset_prefix
      }
    ]
```

```
  })
}
```

You‘re now good to go ! Run and deploy your infrastructure:

```
terraform init
terraform apply
```

The AWS Batch infrastructure will be ready to process your documents, stored in the S3 bucket.

```
aws batch submit-job \
  --job-name <YOUR-JOB-NAME> \
  --job-queue demo-job-queue \
  --job-definition demo-job-definition
```

What about the cost?

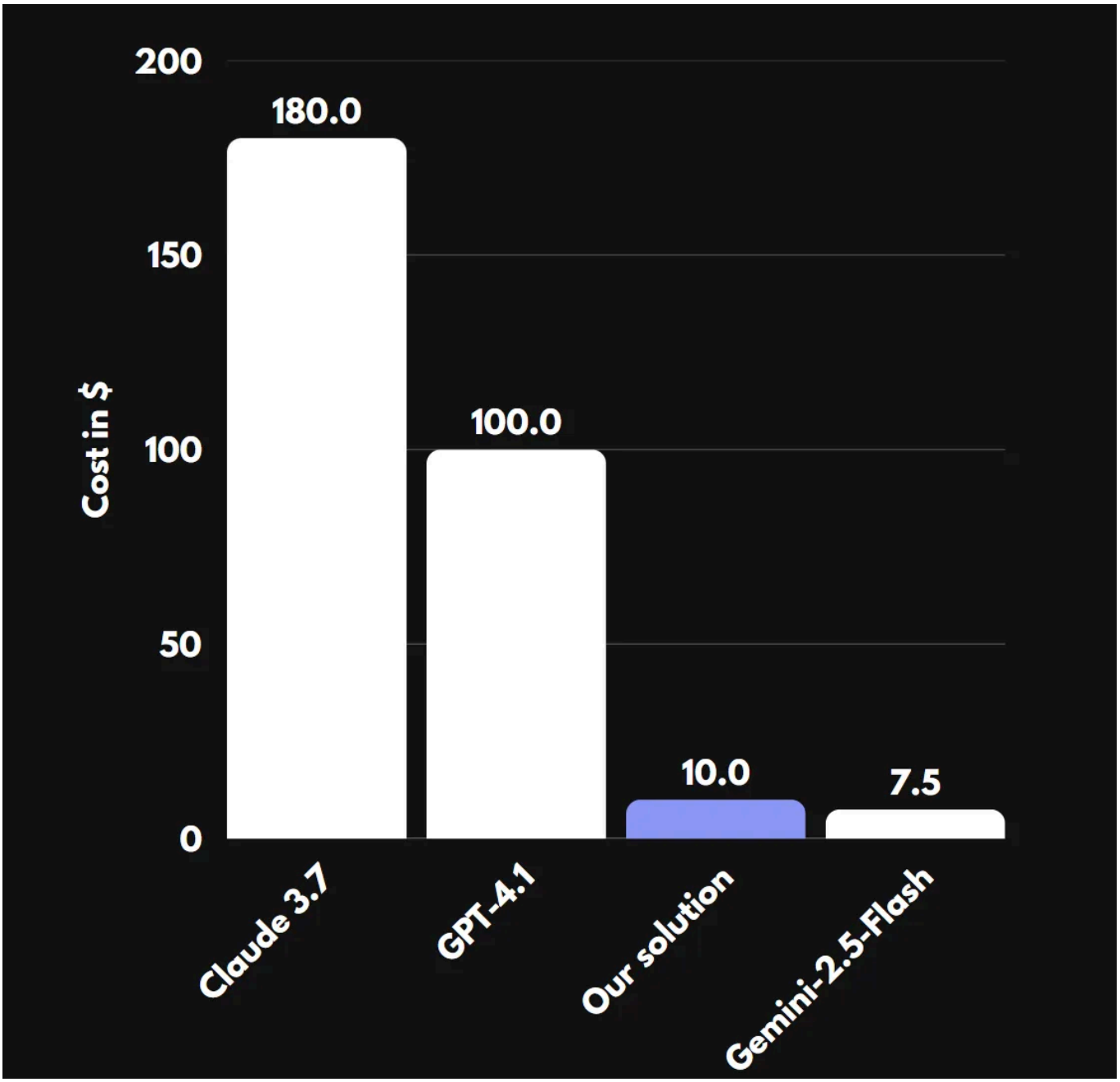
Privacy and reliability are the best advantages for deploying your in-house LLM feature. **But is it that cheap?**

The answer: **yes!**

We did an experimention with 10 documents. It took in average **4.5s per document with vLLM and Qwen-2.5-VL-3B.** A EC2 g6.xlarge instance costs today **$0.8048.**

For 10,000 processed documents, it then represents a cost of **~$10** and would take **~12.5 hours**.

We compare our solution in term of cost to exisiting VLMs such **GPT-4.1, Gemini-2.5-Flash, or Claude 3.7.**



Cost for processing 10,000 documents

This makes our solution not only completly secured, but also one of the most cost efficient.

> *Using an open-source model opens the door for fine-tuning, which improves the performances way further than any general LLM.*

To wrap it up

In this article, we introduce the deployment of an AI feature to parse documents such as invoices or reports. We used **Qwen-2.5-VL** coupled with **vLLM** to predict structured outputs directly from the images. Without any training nor data collection required!

The feature is containerized with **Docker** and **uv,** then deployed for Batch Inference on **AWS Batch** with **Terraform.** We showed how to deploy the Batch Infrastructure with EC2 orchestration to leverage GPUs.

Additionally, developing our feature in-house proved to be the cheapest solution amongst existing LLM providers. Without any inference or deployment optimization, such as quantization or leveraging better resources (Spot instances or more performant GPU instances).

GitHub - jeremyarancio/VLM-Batch-Deployment: Batch Deployment for Document Parsing with AWS Batch &...

Batch Deployment for Document Parsing with AWS Batch & Qwen-2.5-VL - jeremyarancio/VLM-Batch-Deployment

github.com

I hope you liked this article.

It represents everything I wished I knew before delving into this project. Now go deploy your open-source LLM features!

To stay tuned with my latest content, you can follow me on Linkedin or subscribe to my newsletter.

Subscribe to my newsletter

Subscribe to my newsletter I talk about NLP, last trends in AI, ML engineering and Freelancing. By signing up, you will...

medium.com

Happy coding!

NLP Machine Learning Llm AWS Mlops

Published in Towards AI

81K followers · Last published 10 hours ago

Follow

The leading AI community and content platform focused on making AI accessible to all. Check out our new course platform: https://academy.towardsai.net/courses/beginner-to-advanced-llm-dev

Written by Jeremy Arancio

2.3K followers · 128 following

Follow

NLP Engineer & AI-ndependant - I help companies leverage texts using Machine Learning! - Reach out to me at https://linktr.ee/jeremyarancio

Responses (17)

Rsanders

What are your thoughts?

Alican Kilicarslan
Apr 27 (edited)

Very well explained. Thanks. What about the difference between your work and Gemini 2.5 Flash regarding accuracy?

54 2 replies Reply

Val Neyman
Apr 27

Thanks for the article. Your analysis of closed end LLMs is not accurate. When using API they do not use the data for training unless you opt-in and you can set variables to limit variance.

33 2 replies Reply

Parijatshukla
Apr 26

Thanks, Jeremy for sharing such an insightful article. Best aspect of this article being that it covers the end-to-end solution to the given problem.

39 1 reply Reply

See all responses

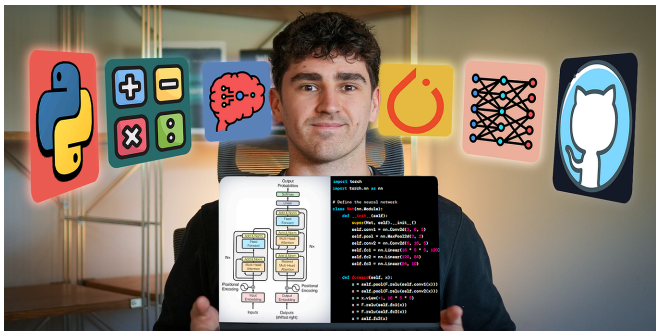
More from Jeremy Arancio and Towards AI

In TDS Archive by Jeremy Arancio

Parse Your Invoices with LayoutLM and Label Studio

Fine-tune LayoutLM on your invoices with the Transformers library, Label Studio, and AWS…

Apr 16, 2024 526 6



In Towards AI by Boris Meinardus

How I'd learn ML in 2025 (if I could start over)

All you need to learn ML in 2025 is a laptop and a list of the steps you must take.

Jan 2 2K 59

In Towards AI by John Loewen, PhD

Having Streamlit Superpowers: The Best GPT-4 Prompts For…

No-fuss prompting for error-free Python Streamlit data viz code

Apr 14 268 3

In TDS Archive by Jeremy Arancio

Build Machine Learning Pipelines with Airflow and Mlflow:…

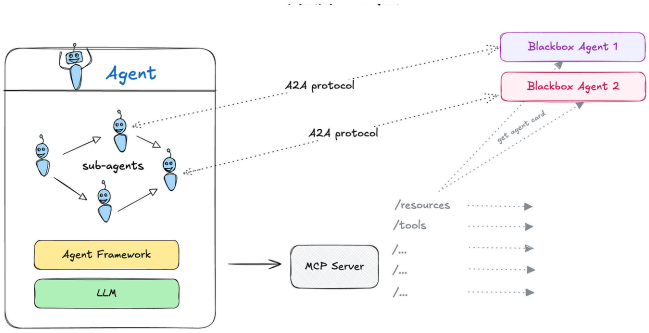
Learn how to create reproducible and ready-for-production Machine Learning pipelines…

Jan 12, 2024 780 6

See all from Jeremy Arancio

See all from Towards AI

Recommended from Medium

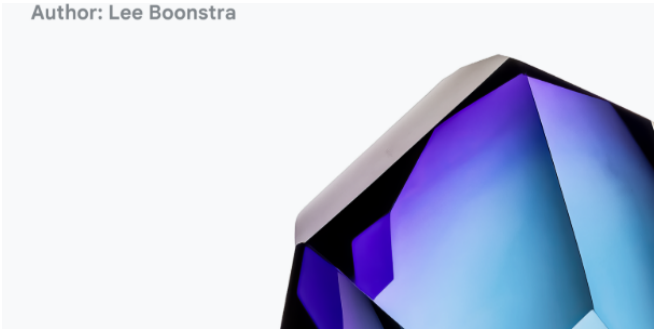


Edwin Lisowski

What Every AI Engineer Should Know About A2A, MCP & ACP

How today's top AI protocols help agents talk, think, and work together

Apr 24 699 15

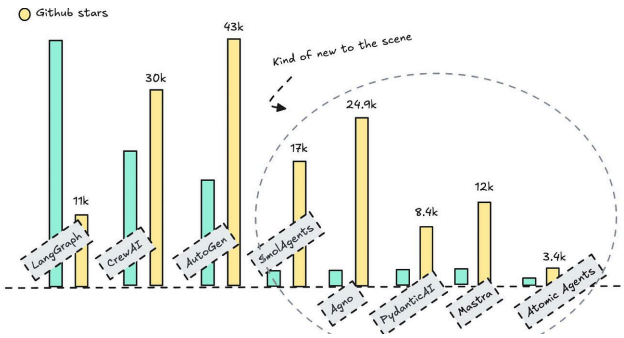


In Coding Nexus by Algo Insights

Google's 69-Page Prompt Engineering Masterclass: What's...

I've been writing about tech for three years, and let me tell you, nothing's grabbed me...

Apr 13 368 11



In Data Science Collective by Ida Silfverskiöld

Agentic AI: Comparing New Open-Source Frameworks

By looking at functionality and learning curves

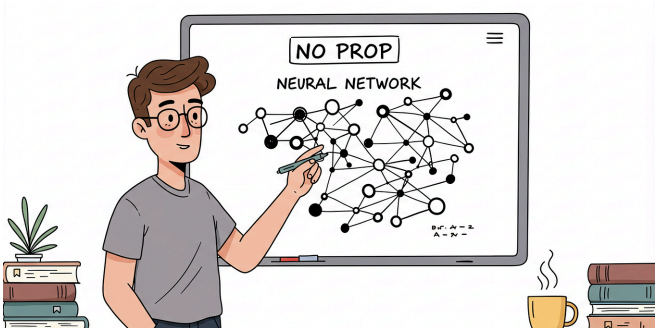
In Level Up Coding by Anmol Baranwal

The guide to MCP I never had

AI agents are finally stepping beyond chat. They are solving multi-step problems,...

Apr 15 1.5K 52

Apr 28 1.1K 16



In AI Advances by Dr. Ashish Bamania

You Don’t Need Backpropagation To Train Neural Networks Anymore

A deep dive into the ‘NoProp’ algorithm that eliminates the need for Forward pass and…

Apr 25 1.5K 14

In ILLUMINATION by Dr. Nikita Singh

After Age 40, You Need to Stop Doing These Morning Habits—…

Your Morning Routine Making You Age Faster?

Apr 21 8.5K 177

See more recommendations