Ray (Task 2)

What is Ray?

An open source framework that helps us to scale ML and Python applications easily. It provides a simple, universal API for building distributed applications that can scale from a laptop to a cluster.

It allows us to scale through distributed computing and Ray simplifies distributed computing by providing:

- **Scalable compute primitives**: Tasks and actors for painless parallel programming
- **Specialized AI libraries**: Tools for common ML workloads like data processing, model training, hyperparameter tuning, and model serving
- **Unified resource management**: Seamless scaling from laptop to cloud with automatic resource handling

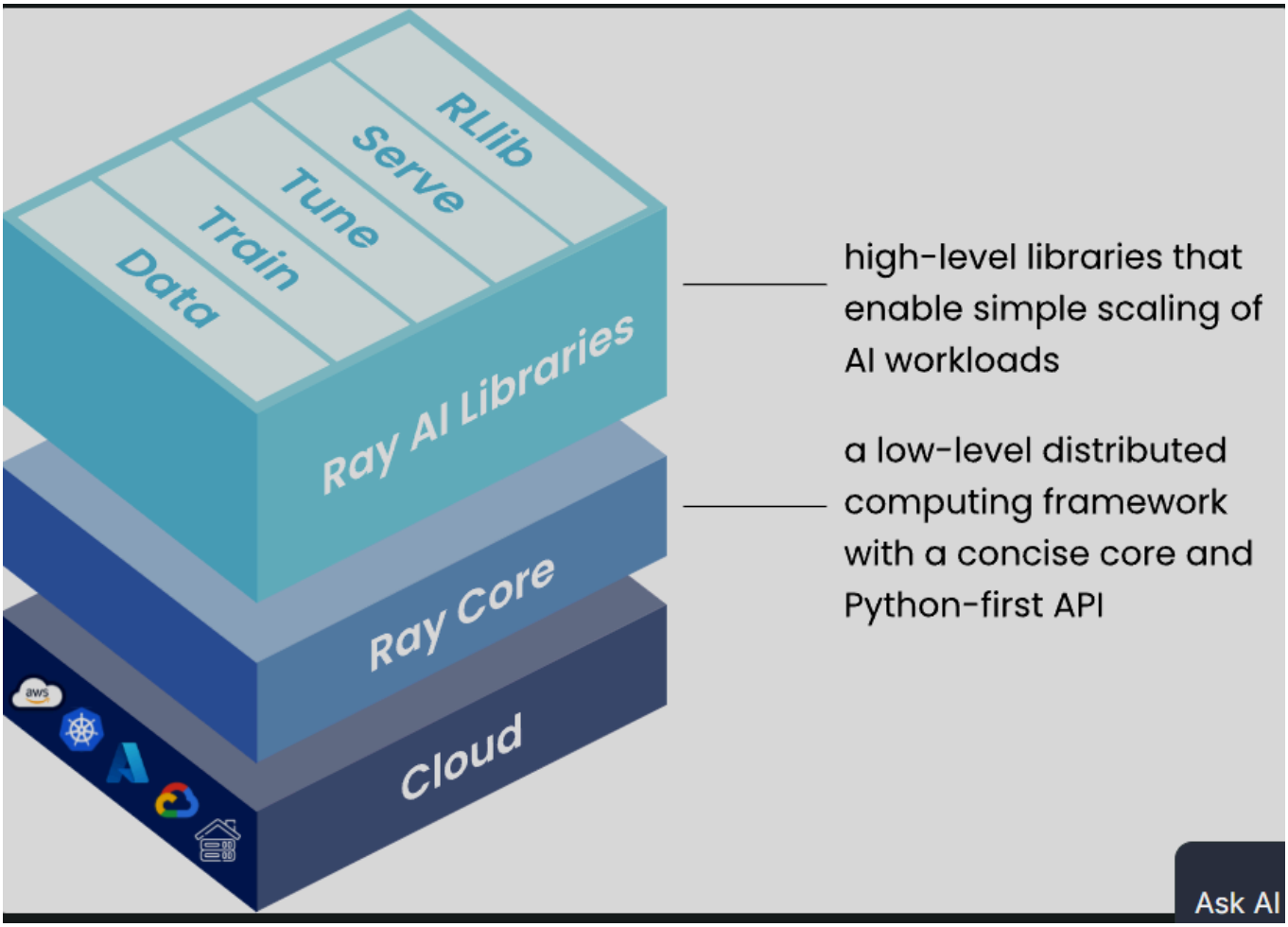
Ray Structure Overview

Ray has two layers:

Layer	Purpose	You use it for
Ray Core	Base engine for distributed execution	Parallel Python tasks, distributed stateful actors
Ray AIR (Ray AI Runtime)	ML and data workload toolkit built on Ray Core	Data pipelines, training, tuning, serving, RL

- **Ray Core** is the **foundation**.
  It provides the **distributed execution engine** (tasks, actors, object store, cluster communication).
- **Ray AI (Ray AIR Runtime)** sits **on top of Ray Core**.
  It provides higher-level **ML-focused libraries**.
- All Ray AI libraries — **Ray Data, Ray Train, Ray Tune, Ray Serve, and RLlib** — **use Ray Core internally** to run distributed workloads.

Our main priority is RAY AI



Ray Core - Walkthrough

Getting Started & Example Usage

This walkthrough gives you a razor-sharp start with Ray Core: initialize a cluster, define tasks & actors, run remote calls, and see the performance boost.

Basic steps

- Install Ray:

```
pip install ray
```

- Initialize Ray:

```
import ray
ray.init()
# You'll see something like: View the Ray dashboard at http://127.0.0.1:8265
```

- Before Ray: sequential example

```
import time

database = [
    "Learning", "Ray",
    "Flexible", "Distributed", "Python", "for", "Machine", "Learning"
]

def retrieve(item):
    time.sleep(item / 10.0)
    return item, database[item]

start = time.time()
data = [retrieve(i) for i in range(len(database))]
print(f"Runtime: {time.time() - start:.2f} seconds, data:")
for d in data:
    print(d)
```

- Using Ray tasks for parallel version

```
import time
import ray

ray.init()

@ray.remote
def retrieve_task(item):
    time.sleep(item / 10.0)
    return item, database[item]

start = time.time()
refs = [retrieve_task.remote(i) for i in range(len(database))]
results = ray.get(refs)
print(f"Runtime: {time.time() - start:.2f} seconds, data:")
for r in results:
    print(r)
```

Notice the drop in runtime because the tasks execute in parallel.

Key takeaway

By following the walkthrough you:

- Start a Ray cluster easily with `ray.init()`.
- Turn ordinary Python functions into distributed tasks with `@ray.remote`.
- Use `.remote()` to launch tasks, and `ray.get(...)` to gather results.
- Share large data via Ray's object store (implicitly or via `ray.put(...)`).

Ray Core - Key concepts

1) Tasks

- Defined by decorating a Python function with `@ray.remote`.
- Executed asynchronously in separate worker processes (possibly on different machines).
- You call the task via `.remote(...)` and immediately get a reference (future) instead of a blocking result.
- Example:

```
import ray
ray.init()

@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(5)]
results = ray.get(futures)
print(results)  # [0, 1, 4, 9, 16]
```

2) Actors

- A stateful worker process created by decorating a class with `@ray.remote`.
- Once created, the actor persists and its methods can be called remotely (via `.remote`).
- Good for managing state across multiple invocations.
- Example:

```
@ray.remote
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
```

```
        self.value += 1
        return self.value

counter = Counter.remote()
print(ray.get(counter.increment.remote()))  # 1
print(ray.get(counter.increment.remote()))  # 2
```

3) Objects / Object Store

- When tasks or actors return values, they are placed in Ray's distributed object store and you get a reference (object ref).
- You can also explicitly use `ray.put(...)` to place an object in the store and obtain a reference.
- These object references can be passed as arguments to other tasks/actors, avoiding unnecessary data copying.
- Example:

```
import numpy as np

matrix = np.ones((1000, 1000))
matrix_ref = ray.put(matrix)
result = ray.get(sum_matrix.remote(matrix_ref))  # if sum_matrix is a remote task
```

4) Placement Groups

- Allow you to reserve a group of resources (like CPUs/GPUs) across nodes in a cluster **atomically**.
- Useful for "gang scheduling" (coordinated scheduling of tasks/actors that must run together).
- Modes: PACK (close together) or SPREAD (spread across nodes).

5) Runtime Environments / Dependencies

- When Ray runs tasks/actors on remote machines, it needs to ensure that required packages, files and environment variables are present.
- You can bundle environment dependencies via Ray's runtime environments or pre-configure the cluster environment.

Key APIs part of RAY AI

- **Ray Data** – Create and transform large datasets efficiently (uses Apache Arrow under the hood)
- **Ray Train** – Run distributed training jobs seamlessly across CPUs/GPUs/nodes
- **Ray Tune** – Run hyperparameter tuning in parallel
- **Ray Serve** – Deploy and serve ML models with autoscaling inference
- **Ray RLlib** – Reinforcement learning algorithms at production scale

Ray Data

Ray Data API is an abstraction of the Ray Dataset API
which represents a distributed collection of data. It is designed specifically for ML workloads and can handle data collections that exceed a single machine's memory.

A little insight on how Ray Data connects with Apache Arrow

Any data format that Arrow supports → Ray can ingest and process efficiently, because Ray uses Arrow as the internal representation for datasets.

Apache Arrow is a columnar in-memory data format toolkit designed for faster data processing across multiple systems.

It is a standard language for data so that different tools Spark, Pandas, Ray, DuckDB, Polars, etc.) can share and process data without converting formats.

Why is this useful?

Normally data gets copied and converted many times:

```
CSV → Pandas → NumPy → Tensor → ML Model
```

Each arrow means **conversion + overhead**.
Apache Arrow allows all of these to share **one efficient memory layout**, so:

```
Arrow table → Pandas / Polars / Ray / Spark / ML frameworks (zero-copy)
```

This means:

- Faster computation
- Less memory usage
- Better interoperability between libraries

Apache Arrow stores data in a columnar format meaning

```
Names: ["John", "Anna", ...]
```

```
Ages:  [  23 ,   31 , ...]
```

instead of traditional row based data storage

```
| Name | Age |
| John | 23  |
| Anna | 31  |
```

Core Capabilities of Ray Data

- Loading data - Ray Data seamlessly integrates with any filesystem supported by Arrow.

```
import ray
# Load a CSV dataset directly from S3
ds = ray.data.read_csv("s3://anonymous@air-example-data/iris.csv")
# Preview the first record
ds.show(limit=1)
```

- Transforming data - Ray takes your user-defined function and applies it to different chunks of your data in parallel, using multiple CPU cores or multiple machines — so the transformation finishes much faster.

```
from typing import Dict
import numpy as np

# Define a transformation to compute a "petal area" attribute
def transform_batch(batch: Dict[str, np.ndarray]) -> Dict[str, np.ndarray]:
    vec_a = batch["petal length (cm)"]
    vec_b = batch["petal width (cm)"]
    batch["petal area (cm^2)"] = vec_a * vec_b
    return batch

# Apply the transformation to our dataset
transformed_ds = ds.map_batches(transform_batch)

# View the updated schema with the new column
# .materialize() will execute all the lazy transformations and
# materialize the dataset into object store memory
print(transformed_ds.materialize())
```

- Consuming data - Access dataset contents through convenient methods like `take_batch()` and `iter_batches()`. You can also pass datasets directly to Ray Tasks

or Actors for distributed processing.

```
# Extract the first 3 rows as a batch for processing
print(transformed_ds.take_batch(batch_size=3))
```

- Saving data - Export processed datasets to a variety of formats and storage locations using methods like `write_parquet()`, `write_csv()`, and more

```
import os

# Save the transformed dataset as Parquet files
transformed_ds.write_parquet("/tmp/iris")
# Verify the files were created
print(os.listdir("/tmp/iris"))
```

Ray Train : Scalable model training API

Ray Train is a scalable machine learning library for distributed training and fine-tuning.

Ray Train allows you to scale model training code from a single machine to a cluster of machines in the cloud, and abstracts away the complexities of distributed computing.

It can be used with different frameworks:

- Pytorch
- Tensorflow
- Transformer, etc.

To use Ray Train effectively, you need to understand four main concepts:

1. Training function: A Python function that contains your model training logic. It is a UDF that has the end-to-end model training loop logic. When launching a distributed training job, each worker executes this function. Conventions for it are:
    1. `train_func` is a user-defined function that contains the training code.
    2. `train_func` is passed into the Trainer's `train_loop_per_worker` parameter.

```
def train_func():
    """User-defined training function that runs on each distributed worker
process.

    This function typically contains logic for loading the model,
    loading the dataset, training the model, saving checkpoints,
    and logging metrics.
```

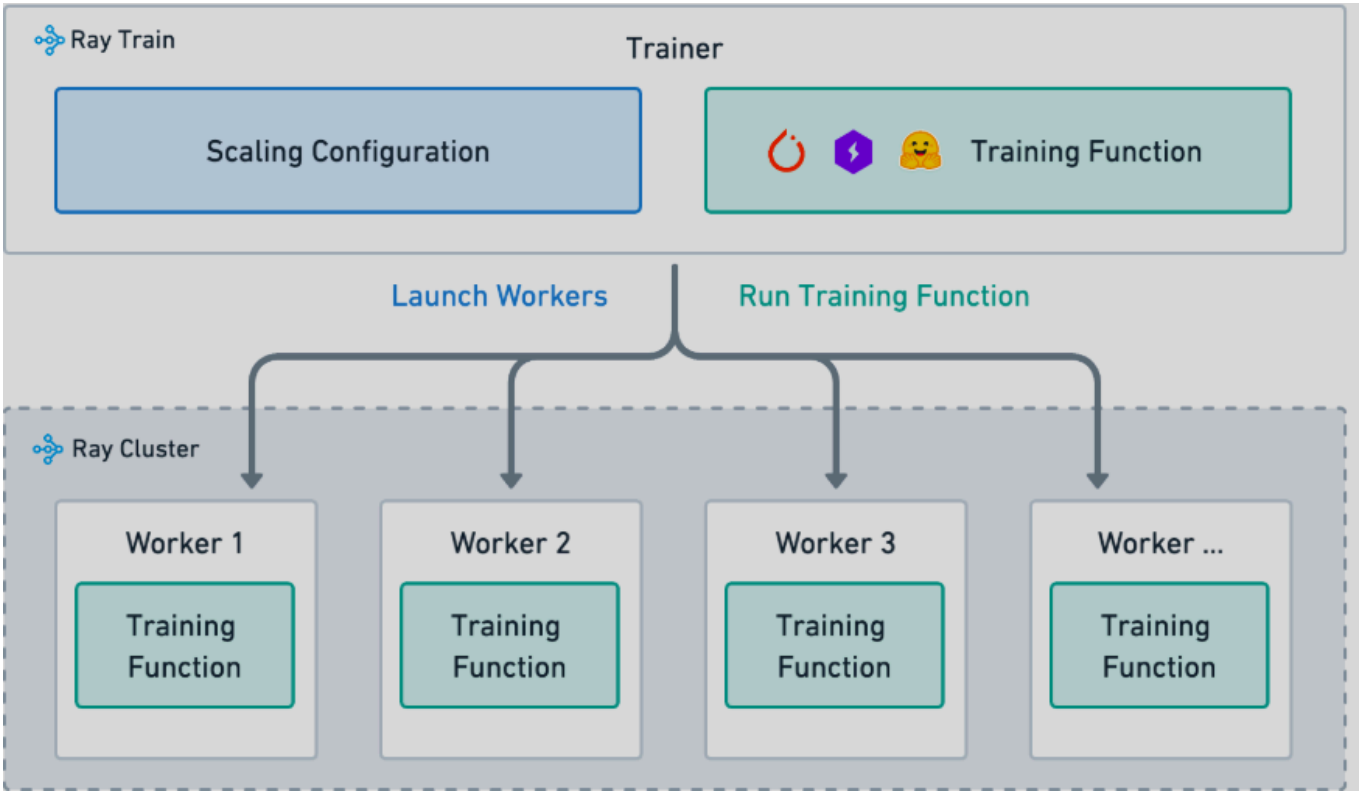
```
    """
```

1. Worker: A process that runs the training function. Each worker is a process that executes the `train_func`. The number of workers determines the parallelism of the training job and is configured in the `ScalingConfig`.
2. Scaling configuration: A configuration of the number of workers and compute resources (for example, CPUs or GPUs). Specify two basic parameters for worker parallelism and compute resources:
    1. `num_workers`: The number of workers to launch for a distributed training job.
    2. `use_gpu`: Whether each worker should use a GPU or CPU.

```
from ray.train import ScalingConfig

# Single worker with a CPU
scaling_config = ScalingConfig(num_workers=1, use_gpu=False)

# Single worker with a GPU
scaling_config = ScalingConfig(num_workers=1, use_gpu=True)

# Multiple workers, each with a GPU
scaling_config = ScalingConfig(num_workers=4, use_gpu=True)
```

4. [Trainer](https://docs.ray.io/en/latest/train/overview.html#train-overview-trainers): A Python class that ties together the training function, workers, and scaling configuration to execute a distributed training job. We call the [`fit()`](https://docs.ray.io/en/latest/train/api/doc/ray.train.trainer.BaseTrainer.fit.html#ray.train.trainer.BaseTrainer.fit "ray.train.trainer.BaseTrainer.fit") method executes the training job.

```python
from ray.train.torch import TorchTrainer

trainer = TorchTrainer(train_func, scaling_config=scaling_config)
trainer.fit()
```



Ray Tune : Scalable hyperparameter tuning

1) Trainable — *The function that runs one training trial*

A **Trainable** is your training logic.
Ray Tune runs it **many times**, each time with a different set of hyperparameters.

You can define it in 2 ways: Functional (shown below) and Class based (less used)

```
from ray import tune

def objective(x, a, b):  # Define an objective function.
    return a * (x**2) + b

def trainable(config):  # Pass a "config" dictionary into your trainable.
    for x in range(20):  # "Train" for 20 iterations and compute intermediate
scores.
        score = objective(x, config["a"], config["b"])
        tune.report({"score": score})  # Send the score to Tune.
```

2) Search Space— *What values to try for each hyperparameter*

This defines the **range** and **sampling strategy** for each hyperparameter.

```
param_space = { "a": tune.uniform(0, 1), "b": tune.randint(0, 10) }
```

Tune provides:

- `tune.choice()` — discrete choices
- `tune.randint()` — integer sampling
- `tune.uniform()` — continuous sampling

3) Trials — *Individual training runs*

Each **Trial** = one model trained with a **specific** hyperparameter configuration.

To launch tuning:

```
tuner = tune.Tuner( trainable, param_space=param_space, ) results = tuner.fit()
```

Tune automatically:

- Schedules trials
- Parallelizes them
- Tracks metrics
- Can stop bad trials early

4) Schedulers — *Optimize resource usage by stopping bad trials early*

Schedulers decide **whether to continue, pause, or stop trials** based on performance.

Example: **ASHAScheduler** (most widely used)

```
from ray.tune.schedulers import ASHAScheduler scheduler = ASHAScheduler()
```

Schedulers reduce cost by focusing on the best-performing trials.

5) Search Algorithms — *Strategies to choose the next hyperparameters to try*

Examples:

- Random search (default)
- Grid search
- **Bayesian optimization (Optuna, HyperOpt)**
- Population-based training

```
from ray.tune.search.optuna import OptunaSearch search_alg = OptunaSearch()
```

Search algorithms improve **efficiency and convergence**.

6) Loggers & Results — *Track and visualize experiment progress*

Tune automatically records:

- Metrics
- Checkpoints
- Configurations
- Trial status

You can view results using:

- Ray CLI dashboard
- TensorBoard
- Weights & Biases / MLflow integration

Summary Table

Concept	Purpose
Trainable	Defines how a single model is trained
Search Space	Defines which hyperparameter values to try
Trials	Actual training runs executed with different configs
Schedulers	Stop/pause/continue trials based on performance
Search Algorithms	Decide what hyperparameters to try next
Loggers & Results	Track trial performance and output best configuration

Ray Serve : # Scalable and Programmable Serving

Ray Serve is a scalable model serving library for builiding online infeerence APIs. Serve is framework agnostic so we can use a single toolkit for everything from deep learning models to normal Python business logic.

Ray Serve is particularly well suited for model composition (where we nest multiple models within each other) and multi-model serving, enabling you to build a complex inference service consisting of multiple ML models and business logic all in Python code.

1. Deployment

A **Deployment** is the core unit of Ray Serve.
It defines *what logic* to execute when a request arrives.

You create deployments using `@serve.deployment`.

```
from ray import serve

@serve.deployment
class MyDeployment:
    def __init__(self, message: str):
        self.message = message

    def __call__(self):
        return self.message

app = MyDeployment.bind("Hello Ray Serve!")
handle = serve.run(app)

assert handle.remote().result() == "Hello Ray Serve!"
```

Each deployment can have one or many **replicas** (workers) that scale based on load.

2. Application

A **Ray Serve Application** is a collection of deployments wired together.
One deployment is designated as the **ingress** (entry point).

```
app = MyDeployment.bind("Hello!")
serve.run(app)
```

Deploying an application starts serving traffic across the cluster.

3. DeploymentHandle & Composition

To call one deployment from another, Ray uses **DeploymentHandles**.

This allows **model composition** and chaining logic.

```
from ray import serve

@serve.deployment
class A:
    def __call__(self):
        return "Hello"

@serve.deployment
```

```python
class B:
    def __call__(self):
        return " world!"

@serve.deployment
class C:
    def __init__(self, a_handle, b_handle):
        self.a = a_handle
        self.b = b_handle

    async def __call__(self):
        return (await self.a.remote()) + (await self.b.remote())

a = A.bind()
b = B.bind()
app = C.bind(a, b)

handle = serve.run(app)
assert handle.remote().result() == "Hello world!"
```

4. Ingress Deployment (HTTP Entry Point)

A deployment can act as the HTTP handler if it receives HTTP requests directly.

```python
import requests
from starlette.requests import Request
from ray import serve

@serve.deployment
class Greeter:
    async def __call__(self, request: Request):
        data = await request.json()
        return f"Hello {data['name']}!"

app = Greeter.bind()
serve.run(app)

assert requests.get("http://127.0.0.1:8000", json={"name": "Ray"}).text ==
"Hello Ray!"
```

FastAPI Integration

Serve integrates cleanly with FastAPI for richer routing.

```
from fastapi import FastAPI
from fastapi.responses import PlainTextResponse
from ray import serve

fastapi_app = FastAPI()

@serve.deployment
@serve.ingress(fastapi_app)
class API:
    @fastapi_app.get("/{name}")
    def greet(self, name: str):
        return PlainTextResponse(f"Hello {name}!")

app = API.bind()
serve.run(app)
```