

MLKit — Generated Project Architecture & Developer Code Specification

This document details the minimal project layout and developer-facing code contracts for generated **MLKit** projects, organized around the three core pillars: **Data**, **Model**, and **Lifecycle**.

1. Project Layout Overview

The standard structure maintains zero hard coupling to specific machine learning runtimes while providing Django-like convention and organization:

```
None
my_ai/
├─ mlkit.json
└─ my_ai/
    ├─ config.py
    ├─ data.py
    ├─ model.py
    ├─ trainer.py
    ├─ evaluator.py
    └─ inference.py
```

File	Role	Pillar / Purpose
mlkit.json	Manifest	Root project metadata and execution intent
config.py	Configuration	Path resolution and environment parameters
data.py	Data	Ingestion, validation, and train/val/test splits
model.py	Model	Model definition, persistence, and forward pass
trainer.py	Lifecycle	Training loops and optimization runs
evaluator.py	Lifecycle	Validation and metric calculation
inference.py	Lifecycle / Serving	Local inference and input handling

2. Manifest: mlkit.json

The root declarative manifest describing the project metadata, runtime configuration, and intent:

```
JSON
{
  "name": "my_ai",
  "version": "0.1.0",
  "type": "classification",
  "entrypoint": "my_ai",
  "config": {
    "device": "auto",
    "batch_size": 32
  }
}
```

3. Configuration: my_ai/config.py

Reads and exposes settings and path anchors from mlkit.json:

```
Python
from pathlib import Path
from modelkit import BaseConfig

BASE_DIR = Path(__file__).resolve().parent.parent

class Config(BaseConfig):
    name: str = "my_ai"
    data_dir: Path = BASE_DIR / "data"
    artifacts_dir: Path = BASE_DIR / "artifacts"
    device: str = "auto"
    batch_size: int = 32
```

4. Data Pillar: my_ai/data.py

Implements data loading, preprocessing, and train/val/test splits using whichever library the developer chooses (e.g., pandas, Hugging Face, PyTorch):

Python

```
import pandas as pd
from modelkit import Dataset

class AIDataset(Dataset):
    def load(self, source=None, **kwargs):
        path = source or (self.config.data_dir / "dataset.csv")
        self.data = pd.read_csv(path)
        return self.data

    def split(self, train=0.8, validation=0.1, test=0.1, **kwargs):
        train_df = self.data.sample(frac=train, random_state=42)
        remaining = self.data.drop(train_df.index)
        val_df = remaining.sample(frac=0.5, random_state=42)
        test_df = remaining.drop(val_df.index)
        return train_df, val_df, test_df
```

5. Model Pillar: my_ai/model.py

Encapsulates model initialization, checkpoint loading/saving, and forward inference without locking into a specific ML backend:

Python

```
import joblib
from modelkit import Model
from sklearn.linear_model import LogisticRegression

class AIModel(Model):
    def __init__(self, name="classifier", config=None):
        super().__init__(name=name, config=config)
        self.instance = LogisticRegression()

    def predict(self, inputs, **kwargs):
        return self.instance.predict(inputs)

    def save(self, destination, **kwargs):
        joblib.dump(self.instance, destination)

    def load(self, source, **kwargs):
        self.instance = joblib.load(source)
```

6. Lifecycle Pillar (Training): my_ai/trainer.py

Orchestrates the model training routine, extracts training subsets, fits the model, and outputs training telemetry:

Python

```
from modelkit import BaseTrainer

class AITrainer(BaseTrainer):
    def fit(self, model, dataset, **kwargs):
        train_data, _, _ = dataset.split()
        X_train = train_data.iloc[:, :-1]
        y_train = train_data.iloc[:, -1]

        model.instance.fit(X_train, y_train)
        return {"status": "success", "samples": len(train_data)}
```

7. Lifecycle Pillar (Evaluation): my_ai/evaluator.py

Calculates validation and test metrics to assess performance:

Python

```
from modelkit import BaseEvaluator
from sklearn.metrics import accuracy_score

class AIEvaluator(BaseEvaluator):
    def evaluate(self, model, dataset, **kwargs):
        _, _, test_data = dataset.split()
        X_test = test_data.iloc[:, :-1]
        y_test = test_data.iloc[:, -1]

        predictions = model.predict(X_test)
        acc = accuracy_score(y_test, predictions)
        return {"accuracy": float(acc)}
```

8. Lifecycle Pillar (Inference & Serving): my_ai/inference.py

Accepts raw incoming payloads, validates inputs, and coordinates output prediction:

Python

```
from modelkit import BaseInference

class AIIInference(BaseInference):
    def run(self, model, raw_input, **kwargs):
        features = [raw_input["features"]]
        predictions = model.predict(features)
        return {"prediction": predictions.tolist()}
```