# FoodVisionMini Project Spec (Training + EvalHarness + Deployment)

## 1) Product Summary

FoodVisionMini is a small, end-to-end computer vision project that proves you can:

1. Train a real image classifier on a real dataset slice.
2. Evaluate it like an engineer (not vibes).
3. Package it for inference.
4. Deploy a demo app people can use.

**User story:** “I upload a food photo and get a prediction with probabilities and the time it took.”

**Core classes (locked for Mini):**

1. pizza
2. steak
3. sushi

**Core deliverables:**

1. A reproducible training pipeline that outputs a saved model weight file.
2. A reusable evaluation harness (EvalHarness) that produces metrics, plots, and failure cases.
3. A Gradio demo app deployable on Hugging Face Spaces (or runnable locally).
4. A clean repo that passes a smoke CI run.

---

## 2) Project Goals (What this project is trying to prove)

### 2.1 Technical goals

1. You can fine-tune a pretrained vision backbone as a feature extractor.
2. You can measure generalization, not just train accuracy.
3. You can measure inference latency on CPU and report it honestly.
4. You can ship a working demo with stable preprocessing and predictable outputs.

### 2.2 Portfolio goals

1. Recruiter can run one command and see results.
2. README tells a full story: data → method → eval → failures → what you would change.
3. Your code is modular enough to scale from Mini (3 classes) to Big (Food101).

**2.3 “Definition of Done”**
FoodVisionMini is “done” when it has:

1. Strong baseline (EfficientNetB2 feature extractor is the baseline).
2. One meaningful improvement (example options in Section 6).
3. Proper evaluation (metrics + plots + slices + latency).
4. Documented failure modes with example images.

---

## 3) Dataset and Data Pipeline

FoodVisionMini uses a **subset of Food101** focused on 3 classes: pizza, steak, sushi.

**3.1 Data layout**
Use a simple folder structure:

1. `data/foodvision_mini/train/pizza/...jpg`
2. `data/foodvision_mini/train/steak/...jpg`
3. `data/foodvision_mini/train/sushi/...jpg`
4. same for `test/` (and optionally `val/` if you split train)

**3.2 Splits**
Pick one and lock it early:

1. Option A (simple): train/test only, no val, but you do cross-validation style sanity checks.
2. Option B (recommended): train/val/test where val is a fixed split from train (stratified by class).

**3.3 Reproducibility rules**

1. Set one SEED in one place and thread it everywhere.
2. Log dataset counts per split per class.
3. Save the exact class order (class_names) and never change it silently.

**3.4 Data transforms**
Use torchvision pretrained transforms for the backbone, because it reduces hidden mismatch risk.

1. For EfficientNetB2: use `EfficientNet_B2_Weights.DEFAULT.transforms()`.
2. If you add augmentation, document it and measure whether it helps or hurts.

**3.5 Data quality checklist (fast but real)**
Run these before training:

1. Count images per class per split.

2. Verify all images open (no corrupted files).
3. Check that labels match folder names.
4. Show a small grid of sample images per class in the README (10 per class is enough).

---

## 4) Model Choices and Why

### 4.1 Baseline model (locked)

EfficientNetB2 feature extractor:

1. Load pretrained EfficientNetB2.
2. Freeze base layers.
3. Replace classifier head with a 3-class head.

This matches your notebook structure (`create_effnetb2_model(num_classes=3)`).

### 4.2 Optional baseline comparison

Vision Transformer (ViT) feature extractor baseline:

1. Same dataset, same split.
2. Same EvalHarness outputs.
3. Compare accuracy and CPU latency.

You only keep this if it teaches you something real.

### 4.3 Why EfficientNetB2 is a good Mini baseline

1. Strong pretrained features.
2. Fast enough on CPU for demos.
3. Easy to package with torchvision transforms.

---

## 5) Training Pipeline Spec

### 5.1 Training entrypoint

Create a script like:

1. `src/train.py` (main entrypoint)
2. `src/models.py` (model constructors)
3. `src/data.py` (datasets, loaders, transforms)
4. `src/config.py` (hyperparams, paths)
5. `src/utils.py` (seed, saving, logging)

**5.2 Training loop requirements**
Your training must produce these artifacts:

1. `artifacts/run_YYYYMMDD_HHMM/`
2. `config.json` (all hyperparams and paths)
3. `metrics.json` (per-epoch train and val metrics)
4. `loss_curve.png` (train vs val loss)
5. `accuracy_curve.png` (train vs val accuracy)
6. `best_model.pth` (weights)
7. `class_names.json` (ordered list)
8. `transform_recipe.json` (human readable transform description)

**5.3 Hyperparameters (initial defaults)**
Start simple and adjust only with evidence:

1. Epochs: 5 to 15
2. Batch size: 32 (adjust if memory limited)
3. Optimizer: Adam or AdamW
4. LR: 1e-3 for head-only training
5. Loss: CrossEntropyLoss
6. Mixed precision: optional, but only if you measure speedups correctly

**5.4 Save rules**

1. Always save the best model by validation accuracy (or val loss if you prefer).
2. Save the last model too, but mark it clearly.
3. Save a single “reproduce command” in the README that rebuilds the same results.

---

## 6) EvalHarness (In Depth) and How FoodVisionMini Uses It

EvalHarness is your reusable evaluation toolkit that makes your projects credible. It is not one function, it is a small set of repeatable evaluation modules that every project can share.

**6.1 Why EvalHarness exists**
Without a harness, each project invents evaluation from scratch, which causes:

1. inconsistent metrics
2. missing robustness tests
3. cherry-picked results
4. unreadable comparisons between runs

EvalHarness makes your results:

1. comparable across models
2. repeatable across weeks
3. defensible to smart reviewers

**6.2 EvalHarness structure (recommended)**
Create `evalharness/` with these modules:

1. `evalharness/metrics_classification.py`
    - accuracy (top-1)
    - top-k accuracy (top-3 for 3 classes is trivial, but include it anyway for template consistency)
    - precision/recall/F1 per class
    - macro and weighted averages
2. `evalharness/confusion.py`
    - confusion matrix
    - normalized confusion matrix
    - “top confusions” list (which class gets confused with what)
3. `evalharness/calibration.py`
    - confidence histogram
    - expected calibration error (ECE) (optional but valuable)
    - reliability diagram (optional)
    - “overconfidence cases” (examples where model is wrong but confident)
4. `evalharness/slicing.py`
    - slice by image resolution buckets
    - slice by brightness buckets
    - slice by aspect ratio buckets
    - slice by predicted confidence buckets
    - output: per-slice metrics table + “worst slice” callout
5. `evalharness/robustness.py`
    - simple corruptions: blur, noise, jpeg compression, random crop
    - measure degradation curves
    - output: “corruption severity vs accuracy” plot
6. `evalharness/perf.py`
    - model size (MB)
    - parameter count
    - CPU inference latency: p50 and p95
    - throughput if you batch
    - include warmup and report whether you warmed up
7. `evalharness/report.py`
    - takes outputs from all modules and writes a single `eval_report.md`
    - saves plots into `artifacts/.../eval/`

**6.3 FoodVisionMini required EvalHarness outputs**
FoodVisionMini must produce, at minimum:

1. Accuracy and per-class F1
2. Confusion matrix image
3. Latency report with p50 and p95 CPU timing
4. At least one robustness test (one corruption sweep)
5. At least one slicing report (brightness or resolution buckets)
6. A short “Failure Cases” section with at least 12 example images:
    - 4 misclassified pizza
    - 4 misclassified steak
    - 4 misclassified sushi
      For each: true label, predicted label, confidence

**6.4 EvalHarness API (simple mental model)**
EvalHarness should look like:

1. `run_eval(model, dataloader, config) -> EvalResults`
2. `save_eval(EvalResults, output_dir)`
3. `make_report(EvalResults, output_dir)`

The goal is: evaluation is one command, not a notebook ritual.

**6.5 What would change my mind statement**
Every FoodVisionMini eval must include:

1. “If these slices fail, I do not trust the model.”
2. “If p95 latency is above X ms on CPU, the demo is not usable.”
3. “If robustness drops by more than Y points under mild corruption, I need better augmentation or preprocessing.”

---

## 7) Improvement Options (Pick 1 Meaningful Improvement)

You only need one improvement beyond baseline, but it must be real.

Pick one:

1. Unfreeze the last block of EfficientNet and fine-tune with a smaller LR, then compare.
2. Add measured augmentation (RandAugment or basic augment set) and show robustness gains.
3. Class-balanced sampling or loss weighting if class distribution is uneven.
4. Test-time augmentation and measure tradeoffs (accuracy up vs latency up).

5. Quantization (dynamic quantization on CPU) and measure latency impact with minimal accuracy loss.

Rule: you must report the tradeoff, not just “it improved.”

---

## 8) Packaging for Inference

### 8.1 What must be saved

1. model weights (`.pth`)
2. class_names in correct order
3. transform recipe
4. inference config (image size, normalization, etc.)

### 8.2 Inference function requirements

A single function should:

1. accept an image
2. apply transforms
3. run model in `eval()` with `torch.inference_mode()`
4. return:
    - dict of class probabilities
    - predicted class
    - elapsed time (seconds or milliseconds)
    - optional: top-3 list

### 8.3 Determinism notes

1. Inference should be deterministic for a fixed input.
2. Timing will vary, but logic should not.

---

## 9) Demo App (Gradio) Spec

This matches the notebook’s `demos/foodvision_mini/app.py` pattern.

### 9.1 App requirements

1. Upload image input.
2. Output:
    - label probabilities (bar chart if available)
    - predicted class
    - inference time

3. Include example images users can click.
4. Must run on CPU.

**9.2 App performance rules**

1. Load model once at startup, not per request.
2. Use `map_location="cpu"` for loading weights.
3. Use `torch.inference_mode()`.

**9.3 App safety rules**

1. Limit input image size or resize early.
2. Do not write uploaded images to disk unless needed.
3. If writing, use a safe temp directory and clean up.

**9.4 Hugging Face Spaces deployment checklist**
Your demo folder must contain:

1. `app.py`
2. `model.py`
3. `requirements.txt` with pinned versions
4. weights file
5. `README.md` describing:
    - what it does
    - model
    - classes
    - how to run locally
    - expected latency

---

## 10) Repo Structure (Clean and Expandable)

Recommended structure:

1. `src/`
    - training + data + utils
2. `evalharness/`
    - shared evaluation modules
3. `demos/foodvision_mini/`
    - `app.py`, `model.py`, `requirements.txt`, weights
4. `artifacts/`
    - run outputs (gitignored, but you can keep a few small example artifacts)
5. `tests/`

    - smoke tests
6. `README.md`
7. `DATA.md`
8. `LICENSE`
9. `environment.yml` or `pyproject.toml`

---

## 11) CI Smoke Tests (Minimal but Real)

Add GitHub Actions that:

1. installs dependencies
2. runs unit tests
3. runs a tiny “smoke train” on a tiny subset (like 8 images per class for 1 epoch)
4. runs EvalHarness on that tiny subset and verifies it produces a metrics JSON

This makes the repo feel professional.

---

## 12) Documentation Requirements (README Must Contain)

FoodVisionMini README must include:

### 12.1 One-paragraph overview

1. what it does
2. what dataset
3. which classes
4. which model
5. what the demo is

### 12.2 Results

1. baseline metrics
2. improvement metrics
3. latency p50 and p95
4. at least one plot

### 12.3 Evaluation details

1. split method
2. metrics definition
3. slices tested

4. robustness tested

**12.4 Failure Modes taxonomy (short but real)**

1. Data failures (lighting, occlusion, weird angles)
2. Objective failures (accuracy hides confidence issues)
3. Model failures (texture bias, background leakage)
4. Optimization failures (overfit, unstable finetuning)
5. Systems failures (slow CPU inference, large model)
6. Human failures (users upload non-food)

**12.5 Reproduce**
One command per:

1. download data
2. train
3. eval
4. run demo

---

## 13) Milestones (Fast, Shippable)

1. Milestone A: baseline training + EvalHarness report + saved weights
2. Milestone B: one improvement + tradeoff report
3. Milestone C: Gradio app running locally
4. Milestone D: deployed on HF Spaces
5. Milestone E: README polish + failure cases gallery

---

## 14) Optional Extensions (Only if You Want More)

1. FoodVisionBig upgrade path (101 classes, same structure)
2. TorchScript or ONNX export and compare latency
3. Quantization-aware training (only if you want to go deeper)
4. Grad-CAM interpretability page (super nice for vision projects)
5. Add a tiny monitoring log (counts, latency, errors) for demo usage

---

If you want, I can turn this into a “seedable” repo blueprint like your quiz spec: exact file list, exact function signatures, and a locked checklist that matches your Doc 1 rules (metric + plot + 5-sentence takeaway per rep, plus failure cases and a reproduce pack).