Module 5: The Deep Learning & Hugging Face Ecosystem

Capstone: A Weather Assistnant – Custome Model Meets Pretrained NLP

Overview

This capstone proves the four claims that anchor Module 5 — and it does it inside a single Python file you can read top to bottom in under thirty seconds:

- A small custom deep-learning model trained on your own domain data is a five-minute project, not a five-month one — and it can outperform a dumb baseline meaningfully on the data you care about.
- A trained model becomes a product only when it is wrapped in input validation, a decision layer, and confidence checks. The wrapper is most of the engineering.
- Pretrained models from Hugging Face slot into the same wrapper architecture with no new patterns. They are functions like any other.
- Combining a custom model with a pretrained one in a single application is the workflow you will use for ninety percent of the AI features you ship in the next five years. This capstone is the simplest example of that combination.

You will start with the trained weather model from Lecture 2 sitting on disk, build a 60-line application around it that accepts natural English questions, classifies the user's intent using a Hugging Face zero-shot model, runs the weather model to make a prediction, applies decision rules, and returns a human-readable answer.

Before You Start

- Have Python 3.10+ and pip installed. Verify with python3 --version.
- Install the runtime dependencies ahead of time: pip install tensorflow scikit-learn pandas numpy joblib transformers torch. The HF model downloads are large; pre-warm the cache by running the script once a few hours before class.
- Have the three artifacts from Lecture 2 in the working directory: weather_model.keras, weather_scaler.pkl, feature_columns.json. If you have not run Lecture 2 yet, do that first.
- Open capstone_weather_assistant.py from https://github.com/AI-Native-Engineer-Course/AI-Native-Engineer/tree/main/module 5/capstone in your editor. The full source is reproduced at the end of this document for reference.
- Have a terminal in the project directory ready for the live run.

- Have the lecture slide decks for Lectures 2, 3, and 4 open on a second screen — you will refer back to specific diagrams during the segments.

During Hands-On — Four Segments at a Glance

- SEGMENT 1 — Load the custom weather model and write the inference function. ~5 min
- SEGMENT 2 — Build the decision layer — four intents, four sets of rules. ~5 min
- SEGMENT 3 — Add the Hugging Face zero-shot intent classifier. ~5 min
- SEGMENT 4 — Wire it all together and run the scripted demo questions live. ~5 min

> **REMEMBER**
> Module 5 is not about training the largest model you can. It is about composing the right ones into something
> one Python script, returning friendly English answers to ordinary questions. That is the workflow you will use

Step 1 — Load the Custom Model, Write the Inference Function

> **WHAT TO LEARN**
> Every ML application starts with three lines: load the model, load the scaler, load the feature manifest. After that, every prediction is a one-liner. Loading happens once at process startup, not per request. This is the same discipline as opening a database connection pool — same principle, same payoff.

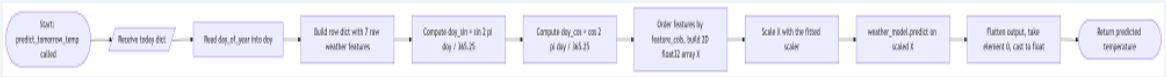
Step 1. Open capstone_weather_assistant.py and walk through Component 1. Three load lines at module scope:

```
weather_model = tf.keras.models.load_model("weather_model.keras")
with open("weather_scaler.pkl", "rb") as f:
    scaler = pickle.load(f)
with open("feature_columns.json") as f:
    feature_cols = json.load(f)
```

Step 2. Then the predict_tomorrow_temp() function. Notice it engineers the cyclical doy features inside the function, because they have to match what the model saw during training:

```
def predict_tomorrow_temp(today: dict) -> float:
    doy = today["day_of_year"]
    row = {
```

```
        **today,
        "doy_sin": float(np.sin(2 * np.pi * doy / 365.25)),
        "doy_cos": float(np.cos(2 * np.pi * doy / 365.25)),
    }
    X = np.array([[row[c] for c in feature_cols]], dtype="float32")
    return float(weather_model.predict(scaler.transform(X), verbose=0).flatten()[0])
```



Flowchart for component 1.

WHAT YOU SHOULD GET

The function returns a single float — tomorrow's predicted max temperature for Orlando, derived from today's error message. The feature manifest is the contract.

Step 2 — Add the Hugging Face Intent Classifier

WHAT TO LEARN

Zero-shot classification is the moment Hugging Face changes how you ship features. The user types "Should I water the lawn tomorrow?" — your code has to know that means "watering intent". Pre-2020 this was a custom-trained classifier requiring labeled examples, a training script, and ongoing maintenance. Today it is one function call against a pretrained model that was never told what your intents are.

Step 1. Walk through Component 2 in the script. The intent classifier is built in two lines using the pipeline API from Lecture 4:

```
intent_classifier = pipeline(
    "zero-shot-classification", model="facebook/bart-large-mnli"
)
```

Step 2. The candidate labels are defined right alongside the model — and they are full descriptive phrases, not single words. This is the prompt-engineering of zero-shot classification:

```
INTENTS = {
    "outdoor activity recommendation": "activity",
    "lawn watering decision":          "watering",
    "heat or temperature warning":     "heat",
    "general weather forecast":        "forecast",
}
```

The keys are what BART sees. The values are the internal symbols your decision layer branches on. Always keep the human-readable labels separate from the application's internal vocabulary.

Step 3. The classify_intent() function does the matching:

```
def classify_intent(user_question: str) -> str:
    result = intent_classifier(
        user_question, candidate_labels=list(INTENTS.keys())
    )
    top_label = result["labels"][0]
    return INTENTS[top_label]
```

Flowchart for component 2.

WHAT YOU SHOULD GET

A function that takes any English sentence and returns one of four internal symbols. Test it live: classify_inter
— generalization for free, no training data required.

Step 3 — The Decision Layer

WHAT TO LEARN

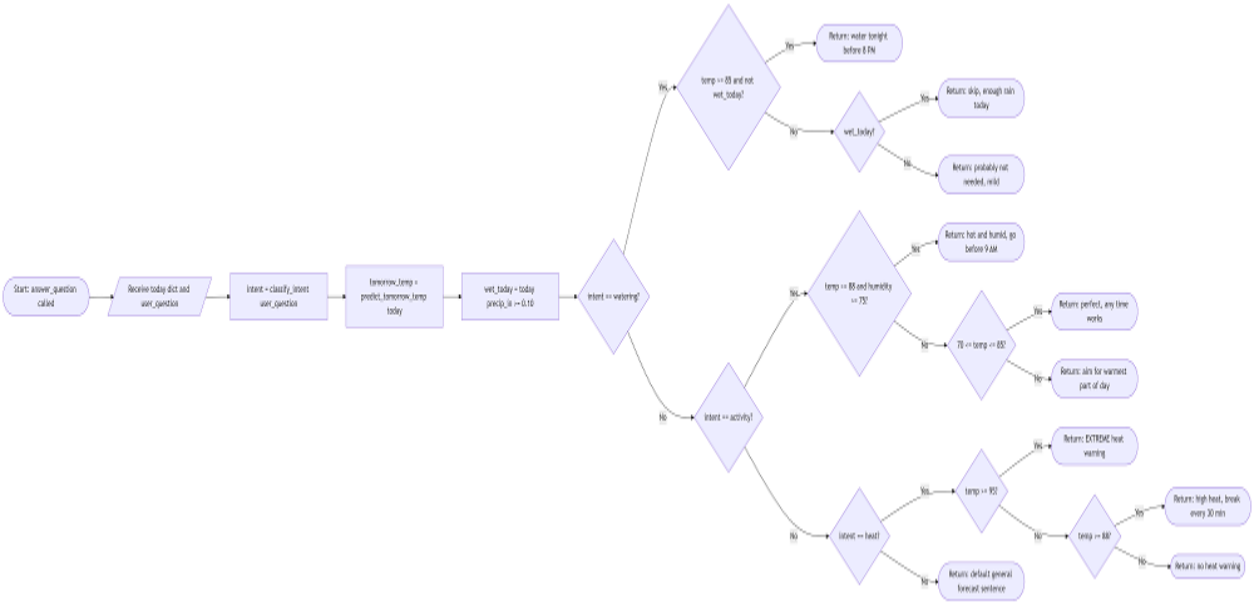
The model produces numbers. Users want answers. The decision layer is where the engineer encodes domain knowledge as auditable rules. Each intent — watering, activity, heat, forecast — maps the predicted temperature into a sentence a person can act on. These rules are NOT learned. They are written by you, in plain Python, and they are the most important code in the entire application because they are what the user actually sees.

Step 1. Walk through the answer_question() function in the script. It branches on intent. Each branch reads like a small policy document:

```
if intent == "watering":
    if tomorrow_temp >= 85 and not wet_today:
        return (f"Yes – tomorrow's predicted high mean temp is
{tomorrow_temp:.1f}°F "
                f"and today was dry. Water this evening before 8 PM.")
    if wet_today:
        return "No need – there's enough rain today to skip watering."
    return f"Probably not needed – tomorrow is mild ({tomorrow_temp:.1f}°F)."
```

Step 2. The activity intent uses two inputs — predicted temperature and humidity — to produce a more nuanced recommendation. Walk through it line by line and stop on the humidity branch:

```
if intent == "activity":
    if tomorrow_temp >= 88 and today["humidity_mean"] >= 75:
        return (f"Tomorrow looks hot and humid ({tomorrow_temp:.1f}°F). "
                f"Plan any outdoor activity before 9 AM.")
    if 70 <= tomorrow_temp <= 85:
        return (f"Perfect outdoor weather tomorrow — predicted {tomorrow_temp:.1f}°F. "
                f"Any time of day works.")
```



Component 3 flowchart.

Step 3. Notice every return is a sentence, not a number. The model returns 91.4; the application says "Tomorrow looks hot and humid." That translation — from float to language — is the application boundary. Numbers stay inside; language comes out.

WHAT YOU SHOULD GET

Four intent branches, each with two to four rules, each returning a complete English sentence. Total: about 4

Step 4 — Wire It Together and Run Live

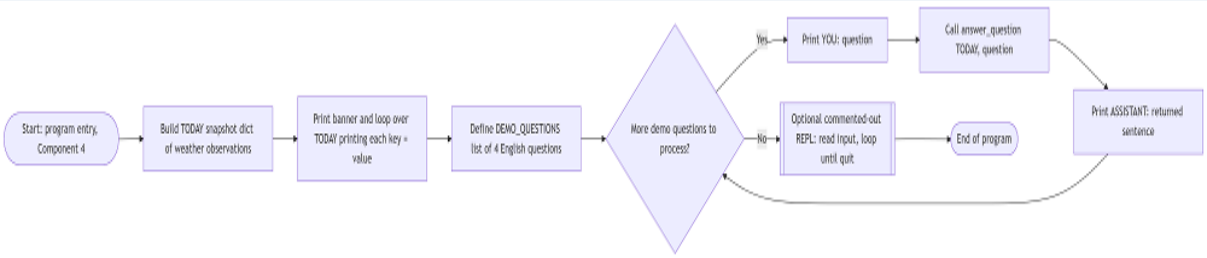
WHAT TO LEARN

The integration step is where many ML demos collapse — too many components, too many state assumptions, too many "but on my machine" surprises. The discipline that keeps it clean is to write a single top-level function that takes raw input and returns raw

> output, and to keep all the wiring inside it. No globals being mutated across calls. No hidden state. Just inputs in, sentences out.

Step 1. Show the answer_question() function in full. It is the orchestrator — three steps, in order:

```
def answer_question(today: dict, user_question: str) -> str:
    intent = classify_intent(user_question)        # 1. HF zero-shot
    tomorrow_temp = predict_tomorrow_temp(today)   # 2. Custom model
    wet_today = today["precip_in"] >= 0.10         # 3. Decision rules
    if intent == "watering": ...
    if intent == "activity": ...
    if intent == "heat":     ...
    return f"My deep learning model predicts ..."  # default / forecast
```



Component 4 flowchart.

Three lines of orchestration. One pretrained model, one custom model, one set of rules. That is the whole application.

Step 2. Run the script. The scripted demo questions execute in order:

```
python3 capstone_weather_assistant.py
```

Watch the four questions print and the four answers stream out. Each one routes through the right intent, runs the weather model once, and produces a complete English response. Total wall-clock time: about three seconds after the models are loaded.

Step 3 (optional, if you have time). Uncomment the REPL block at the bottom of the file. Now the demo is interactive — type a question, get an answer, repeat. Take one question from the room. Watch the model handle it.

> **WHAT YOU SHOULD GET**
>
> Four scripted Q&A pairs printed cleanly. Each answer references the model's actual numeric prediction. The watering and activity answers branch correctly on the today snapshot (humidity 78, no rain). The heat answer responds to the predicted temperature.

The forecast answer is the default fallback that always works. This is a complete AI feature, deployable behind any HTTP server with about ten more lines of code.

Wrap-Up — One File, Two Models, A Feature That Ships

REMEMBER

In 20 minutes you ran a custom Keras model alongside a billion-parameter pretrained BART model inside a single Python file, gave the user a natural-language interface, and produced auditable, rule-based decisions on top of both. That is not a toy. That is the actual workflow you will use to ship the next ten AI features at your company. The hard part was never the model. The hard part is the wrapper — and you have just written one.

Key takeaways:

- Custom + pretrained is the dominant integration pattern. Train what you must, download what you can.
- Load both models once at module scope. Keep call sites cheap and stateless.
- Zero-shot classification eliminates the need to maintain a custom intent classifier for small label sets. Pick descriptive labels; let BART do the rest.
- The decision layer is where domain knowledge lives. Keep it in plain Python, keep it auditable, keep it under version control.
- Numbers stay inside; sentences come out. The application boundary is where floats become language. Crossing that boundary correctly is the engineering work.

STUDENT LAB — MODULE 5 CAPSTONE SUBMISSION

Take capstone_weather_assistant.py and extend it with one additional intent of your own choosing — examples: "best running time", "should I wash my car", "should I plan an indoor day for the kids". Your submission must include five things:

- A new intent string and internal symbol added to the INTENTS dictionary.
- A new branch in answer_question() with at least three rules that consult the predicted temperature, humidity, or precipitation.
- Two test sentences that demonstrate the new intent classifying correctly via zero-shot.
- A short README section (5–10 lines) documenting what your new intent does and what the rules are.

- A 2–3 minute screen recording of you running the script and asking it your two test sentences, with the answers printed to terminal.

This is the gate to Module 6 — Local LLMs and RAG architectures, where you will replace the BART model with a locally hosted open-source LLM and add retrieval over your own documents.

Appendix — capstone_weather_assistant.py (full source)

Save this file as capstone_weather_assistant.py in the same directory as weather_model.keras, weather_scaler.pkl, and feature_columns.json. The script is intentionally a single file so the class can read it top-to-bottom during the live demo.

```
"""
capstone_weather_assistant.py

Module 5 Capstone Application: a Weather Assistant that ties together
every concept from the module —
    - A custom trained deep-learning model (Lecture 2)
    - A decision/inference layer that turns predictions into advice (Lecture 3)
    - Pretrained Hugging Face models that let the user ask questions
      in natural English (Lecture 4)

User flow:
    User types:  "Should I water the lawn tomorrow?"
    Assistant:   classifies intent (zero-shot) → runs the weather model
                 → applies decision rules → returns a friendly answer.

Prerequisites — run in this order before launching the assistant:
    1. python generate_weather_data.py        # creates orlando_weather.csv
    2. python lecture2_train_weather_model.py # creates weather_model.keras +
scaler
    3. pip install transformers torch          # for the HF pipeline
    4. python capstone_weather_assistant.py

This is intentionally a single file so the class can read it top-to-bottom
during the live demo.
"""

import json
import pickle
import numpy as np
import tensorflow as tf
```

```
from transformers import pipeline


# ============================================================================
# Component 1 — load the custom trained model
# ============================================================================
print("Loading custom weather model ...")
weather_model = tf.keras.models.load_model("weather_model.keras")
with open("weather_scaler.pkl", "rb") as f:
    scaler = pickle.load(f)
with open("feature_columns.json") as f:
    feature_cols = json.load(f)


def predict_tomorrow_temp(today: dict) -> float:
    doy = today["day_of_year"]
    row = {
        "temp_max_f":        today["temp_max_f"],
        "temp_min_f":        today["temp_min_f"],
        "temp_mean_f":       today["temp_mean_f"],
        "precip_in":         today["precip_in"],
        "wind_max_mph":      today["wind_max_mph"],
        "humidity_mean":     today["humidity_mean"],
        "pressure_mean_hpa": today["pressure_mean_hpa"],
        "doy_sin":           float(np.sin(2 * np.pi * doy / 365.25)),
        "doy_cos":           float(np.cos(2 * np.pi * doy / 365.25)),
    }
    X = np.array([[row[c] for c in feature_cols]], dtype="float32")
    return float(weather_model.predict(scaler.transform(X),
verbose=0).flatten()[0])


# ============================================================================
# Component 2 — load Hugging Face NLP for intent classification
# ============================================================================
print("Loading Hugging Face intent classifier ...")
intent_classifier = pipeline(
    "zero-shot-classification", model="facebook/bart-large-mnli"
)

INTENTS = {
    "outdoor activity recommendation": "activity",
    "lawn watering decision":          "watering",
    "heat or temperature warning":     "heat",
    "general weather forecast":        "forecast",
}


def classify_intent(user_question: str) -> str:
```

```
    """Map free-form English to one of the four supported intents."""
    result = intent_classifier(user_question,
candidate_labels=list(INTENTS.keys()))
    top_label = result["labels"][0]
    return INTENTS[top_label]


# ============================================================================
# Component 3 — decision layer
# ============================================================================
def answer_question(today: dict, user_question: str) -> str:
    intent = classify_intent(user_question)
    tomorrow_temp = predict_tomorrow_temp(today)
    wet_today = today["precip_in"] >= 0.10

    if intent == "watering":
        if tomorrow_temp >= 85 and not wet_today:
            return (
                f"Yes — tomorrow's predicted high mean temp is
{tomorrow_temp:.1f}°F "
                f"and today was dry. Water this evening before 8 PM."
            )
        if wet_today:
            return "No need — there's enough rain today to skip watering."
        return f"Probably not needed — tomorrow is mild ({tomorrow_temp:.1f}°F)."

    if intent == "activity":
        if tomorrow_temp >= 88 and today["humidity_mean"] >= 75:
            return (
                f"Tomorrow looks hot and humid ({tomorrow_temp:.1f}°F). "
                f"Plan any outdoor activity before 9 AM."
            )
        if 70 <= tomorrow_temp <= 85:
            return (
                f"Perfect outdoor weather tomorrow — predicted
{tomorrow_temp:.1f}°F. "
                f"Any time of day works."
            )
        return (
            f"Tomorrow will feel like {tomorrow_temp:.1f}°F. "
            f"Aim for the warmest part of the day if you're going outside."
        )

    if intent == "heat":
        if tomorrow_temp >= 95:
            return (
                f"EXTREME heat warning: tomorrow's predicted mean is
{tomorrow_temp:.1f}°F. "
                f"Limit outdoor exposure and hydrate aggressively."
```

```
            )
        if tomorrow_temp >= 88:
            return (
                f"High heat tomorrow ({tomorrow_temp:.1f}°F). "
                f"Take a 5-minute break every 30 minutes if working outside."
            )
        return f"No heat warning — tomorrow predicted around
{tomorrow_temp:.1f}°F."

    # default = general forecast
    return (
        f"My deep learning model predicts tomorrow's mean temperature in Orlando
"
        f"will be about {tomorrow_temp:.1f}°F."
    )


# ============================================================================
# Component 4 — interactive demo
# ============================================================================
TODAY = dict(
    temp_max_f=91, temp_min_f=74, temp_mean_f=82,
    precip_in=0.0, wind_max_mph=9, humidity_mean=78,
    pressure_mean_hpa=1014, day_of_year=200,
)

print("\n" + "=" * 60)
print("WEATHER ASSISTANT — ask anything about tomorrow's weather")
print("Today's snapshot:")
for k, v in TODAY.items():
    print(f"   {k:18s} = {v}")
print("Type 'quit' to exit.")
print("=" * 60 + "\n")

DEMO_QUESTIONS = [
    "Should I water the lawn tomorrow?",
    "Is it safe for my kids to play soccer outside in the afternoon?",
    "Will it be dangerously hot tomorrow?",
    "What's the weather going to be like tomorrow?",
]

print("Running scripted demo questions first ...\n")
for q in DEMO_QUESTIONS:
    print(f"YOU: {q}")
    print(f"ASSISTANT: {answer_question(TODAY, q)}\n")

# Optional interactive REPL — uncomment if you have time live
# while True:
#     q = input("YOU: ").strip()
```

```
#     if q.lower() in {"quit", "exit", "q"}:
#         break
#     print(f"ASSISTANT: {answer_question(TODAY, q)}\n")
```