**Phase 0: The Setup (First 30 Minutes)**

**Goal:** Everyone has a identical, working environment. No time wasted on this later.

* **Task:**
  1. Create a shared Git repository.
  2. Create a requirements.txt file with: streamlit, torch, onnx, onnxruntime, scikit-learn, numpy, pandas, plotly.
  3. Create a project structure:
  4. /project
  5. |-- /models\_onnx # Download MobileNet/ResNet/SqueezeNet here
  6. |-- /rtl\_templates # Place your Verilog template here
  7. |-- /trained\_models # Saved AI models will go here
  8. |-- /data # Synthetic data will be saved here
  9. |-- app.py # Main Streamlit app
  10. |-- core\_parser.py # Module 1
  11. |-- core\_predictor.py # Module 2 & 7
  12. |-- core\_search.py # Module 3 & 8
  13. |-- core\_rtl.py # Module 5
  14. |-- train\_ai.py # Module 6
  15. |-- requirements.txt
  16. Everyone runs pip install -r requirements.txt.
* **Success Metric:** Everyone can run streamlit run app.py and see a blank page without any import errors.

**Phase 1: The Core Pipeline - "Dumb" but Functional (Hours 1-4)**

**Goal:** Create a full E2E pipeline that takes an ONNX file and produces a recommendation using simple formulas, NOT AI. This is your safety net.

**Module 1: ONNX Parser & Feature Extractor**

* **Context for LLM Agent:** We need to analyze a machine learning model file in the ONNX format. The goal is to extract its architecture layer by layer, calculating key metrics like the number of parameters and Floating Point Operations (FLOPs). This information will be structured into a JSON object for other parts of our system to use. This is the foundation of our entire analysis.
* **Task for LLM Agent (in core\_parser.py):**
  + Create a Python function parse\_onnx(onnx\_model\_path).
  + It should use the onnx library to load the model.
  + Iterate through the model's graph nodes. For each Conv layer, calculate its parameters and FLOPs using the formula: FLOPs = 2 \* Cin \* K^2 \* Cout \* Hout \* Wout. Handle other layer types gracefully (e.g., ReLU, Pool).
  + Return a dictionary with a specific structure:

JSON

{

"model\_name": "MobileNetV2",

"total\_flops": 300e6,

"total\_params": 3.5e6,

"layers": [

{"layer\_name": "conv1", "type": "Conv", "flops": 1.2e6, "params": 864},

...

]

}

* **Success Metric:** The function runs on a real mobilenetv2-7.onnx file without crashing and outputs a JSON with plausible numbers.

**Module 2: The "Dumb" Performance Predictor**

* **Context for LLM Agent:** This module acts as a simple, formula-based simulator. It takes the model's characteristics (from Module 1) and a hypothetical hardware configuration. Its job is to *estimate* performance metrics like latency and power without running any real hardware. This allows for rapid "what-if" analysis.
* **Task for LLM Agent (in core\_predictor.py):**
  + Create a function predict\_performance\_simple(model\_json, hardware\_config).
  + The hardware\_config is a dict: {'array\_size': 16, 'precision': 'INT8', 'clock\_ghz': 1.0}.
  + Implement these formulas:
    - effective\_flops\_per\_second = (array\_size \* array\_size) \* clock\_ghz \* 1e9 \* 2 (2 ops per MAC)
    - predicted\_latency\_ms = (model\_json['total\_flops'] / effective\_flops\_per\_second) \* 1000
    - predicted\_power\_w = model\_json['total\_flops'] \* 1e-9 (A placeholder, e.g., 1W per GFLOP)
  + Return a dictionary: {'latency\_ms': 25.5, 'power\_w': 4.1}.
* **Success Metric:** The function takes the output from Module 1 and returns a performance dictionary with calculated values.

**Module 3: The Simple Search Loop**

* **Context for LLM Agent:** This module is the "brain" of the operation, in its simplest form. It orchestrates the search for the best hardware configuration. It will generate a list of possible hardware configurations, use the predictor (Module 2) to evaluate each one, and then filter and rank them based on user-defined constraints.
* **Task for LLM Agent (in core\_search.py):**
  + Create a function run\_simple\_search(model\_json, constraints).
  + The constraints is a dict: {'max\_latency\_ms': 50, 'max\_power\_w': 5}.
  + Hardcode a list of hardware configs to test (e.g., array\_size from 4 to 16, precision as 'INT8'/'FP16').
  + Loop through each config, call predict\_performance\_simple from Module 2.
  + Store the results that meet the user's constraints.
  + Return a sorted list of the top 3 best-performing valid configurations.
* **Success Metric:** The function correctly filters out configs that violate constraints and returns a ranked list.

**Phase 2: UI and Tangible Outputs (Hours 5-7)**

**Goal:** Wrap the "Golden Path" pipeline in a usable web interface and generate the final Verilog output.

**Module 4: The Streamlit UI**

* **Context for LLM Agent:** This is the user-facing front-end. We will use the Streamlit library to create a simple web application. The UI needs a file uploader for the ONNX model, sliders for the user to input their performance constraints, a button to trigger the analysis, and areas to display the results (tables and graphs).
* **Task for LLM Agent (in app.py):**
  + Set up a title: "AI-Powered Neural Accelerator Optimizer".
  + Create a file uploader for .onnx files.
  + Create sliders for "Max Latency (ms)", "Max Power (W)".
  + Create a "Run Optimization" button.
  + When the button is clicked:
    1. Call the parser from core\_parser.py (Module 1).
    2. Call the search function from core\_search.py (Module 3).
    3. Display the results in a st.table().
    4. Use plotly to create a scatter plot of latency vs. power for all tested configurations, highlighting the recommended ones.
* **Success Metric:** You can upload a model, set constraints, click the button, and see a table and plot of results. The E2E "dumb" pipeline is now fully interactive.

**Module 5: The Verilog RTL Generator**

* **Context for LLM Agent:** The final output of our tool is a hardware design file in the Verilog language. We will not be writing a complex compiler. Instead, we will use a pre-written template-based approach. The task is to create a Python function that takes a recommended hardware configuration and populates a Verilog template file with the specific parameters.
* **Task for LLM Agent (in core\_rtl.py):**
  + Create a Verilog template file rtl\_templates/systolic\_array\_template.v. This file should contain placeholders like parameter ARRAY\_SIZE = \_\_ARRAY\_SIZE\_\_; and parameter DATA\_WIDTH = \_\_DATA\_WIDTH\_\_;.
  + Create a Python function generate\_verilog(config\_dict).
  + The function should read the template file as a string.
  + It should replace the placeholders with values from the config\_dict (e.g., \_\_ARRAY\_SIZE\_\_ becomes 12).
  + Return the final Verilog code as a string.
  + **In app.py**: Add a download button (st.download\_button) to save the generated Verilog file.
* **Success Metric:** The UI shows a download button that saves a valid .v file with the correct parameters from the recommended configuration.

**Phase 3: The AI Upgrade (Hours 8-11)**

**Goal: Replace the "dumb" placeholder modules (2 & 3) with the advanced, research-grade AI core described in your text. This phase is divided into three parallelizable tasks.**

**Module 6: Data Synthesis & Augmentation**

**Context for LLM Agent: Before we can train our advanced AI models, we need a high-quality dataset. This module's task is to create this dataset. It will generate a base of 300-800 "synthetic" samples using mathematical formulas seeded from real-world hardware benchmarks (like HW-NAS-Bench). It will then "augment" this with about 100 real performance measurements taken from running actual models on an RTX 4060 GPU. The final output is a single CSV file that will be the "ground truth" for training our AI.**

**Task for LLM Agent (in train\_ai.py):**

1. **Synthesize Data:**
   * **Create a function synthesize\_data(num\_samples=500).**
   * **Create a loop to generate num\_samples. In each loop, randomly generate inputs: total\_flops (e.g., 1e9 to 50e9), total\_params (e.g., 1e6 to 100e6), array\_size (4-16), precision (8, 16, 32), batch\_size (1-16).**
   * **For each generated sample, calculate the "true" performance using the formulas you provided:**
     + **latency = (flops \* sparsity\_factor) / (array\_size\*\*2 \* clock\_speed \* precision\_factor)**
     + **memory = (params \* (precision / 8)) + buffer\_overhead**
     + **power = (flops \* power\_coefficient)**
   * **Add random noise (numpy.random.normal) to the calculated outputs to make the data more realistic.**
   * **Return a Pandas DataFrame with all this data.**
2. **Augment with Real Data (Optional but Recommended):**
   * **Create a function run\_real\_benchmarks(onnx\_models\_list).**
   * **This function will loop through a few real ONNX models, run inference on them using onnxruntime-gpu, and measure the actual latency and memory usage. This is the most challenging part; you can start with a simplified version.**
3. **Combine and Save:**
   * **In the if \_\_name\_\_ == '\_\_main\_\_': block, call the synthesis function.**
   * **(If implemented) Call the benchmark function and merge the results.**
   * **Save the final combined DataFrame to /data/training\_dataset.csv.**

**Success Metric: A CSV file named training\_dataset.csv exists in the /data folder with 500+ rows of clean, numerical data.**

**Module 7: The "Smart" Predictor (MoE Implementation)**

**Context for LLM Agent: This module will replace our simple formula-based predictor. The task is to build and train a Mixture of Experts (MoE) model using PyTorch. The MoE will have a "gating network" that routes inputs to one of four specialized "expert" models. The experts themselves can be simple scikit-learn regressors. This MoE will be trained on the dataset created in Module 6. The final trained models (gating and experts) should be saved to disk.**

**Task for LLM Agent (continue in train\_ai.py):**

1. **Load Data: Load the training\_dataset.csv from Module 6. Split it into features (X) and multiple targets (y\_latency, y\_power, y\_memory, y\_throughput).**
2. **Implement Static Gating: The "gating" network can be a simple classifier (like LogisticRegression or a small PyTorch nn.Module) that predicts which expert is best for a given input. For "static soft gating," you'll use softmax on the output logits to get routing weights.**
3. **Train the Experts:**
   * **Train four separate scikit-learn regressor models (e.g., RandomForestRegressor or GradientBoostingRegressor), one for each target: latency, power, memory, and throughput. These are your "experts."**
4. **Save the Models: Save the trained gating model and all four expert models to the /trained\_models directory using joblib or pickle.**

**Success Metric: You have multiple .pkl files in the /trained\_models folder, representing your complete MoE system.**

**Module 8: The "Smarter" Search (Federated Meta-NAS)**

**Context for LLM Agent: This module replaces the simple brute-force search loop. The goal is to implement an advanced Neural Architecture Search (NAS) algorithm. This algorithm will learn how to intelligently explore the hardware configuration space. It will use the MoE from Module 7 as its performance evaluator. We will simulate "Federated Meta-Learning" by pretending to train the NAS searcher on data split across different machines.**

**Task for LLM Agent (in core\_search.py):**

1. **Create a New Search Function: Create a new function run\_nas\_search(model\_json, constraints).**
2. **Load the Predictor: Inside this function, load the trained MoE models from Module 7. This will be your predictor function.**
3. **Implement NAS Logic:**
   * **This is the most complex part. A simple "evolutionary search" is a great starting point and fits the NAS concept.**
   * **Initialization: Create a "population" of 10 random hardware configurations.**
   * **Evaluation Loop (20-30 iterations):**
     + **For each configuration in the population, use the MoE predictor to get its performance score (its "fitness").**
     + **Selection: Keep the top 5 "fittest" configurations that meet the user's constraints.**
     + **Mutation/Crossover: Create 5 new "offspring" configurations by slightly changing the parameters of the best ones (e.g., change array size from 8 to 12).**
     + **Replace the old population with this new, improved one.**
4. **Simulate Federation (The "Talking Point"): The "federated" aspect is a conceptual simulation. You can demonstrate this by having your training script (Module 6 & 7) optionally load only a *slice* of the dataset, mimicking how it would train on a local data split in a real federated setup. This is more for the presentation than for the code itself.**
5. **Return the Result: After the loop finishes, return the best configuration found.**

**Success Metric: The run\_nas\_search function returns a valid hardware configuration that is better (or found faster) than the result from the simple search. The function correctly uses the loaded MoE models to guide its search.**