**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 AI core. This is where you win.

**Module 6: Data Synthesis & MoE Training**

* **Context for LLM Agent:** To train our AI predictors, we need data. This script will generate a synthetic dataset that mimics real-world hardware performance. It will then train a Mixture of Experts (MoE) model. An MoE model uses a "gating network" to route an input to one of several specialized "expert" models. Here, we'll have experts for latency, power, etc.
* **Task for LLM Agent (in train\_ai.py):**
  + Generate a Pandas DataFrame with ~500 rows. Columns should be model\_flops, model\_params, array\_size, precision\_val, etc., and the target columns latency, power. Generate these with formulas plus numpy.random noise to make it realistic.
  + Implement a simple MoE structure using scikit-learn. The "gating" can be a KNeighborsClassifier and the "experts" can be RandomForestRegressor models (one for latency, one for power).
  + Train the gating network and the expert regressors on the synthetic data.
  + Save the trained models to the /trained\_models directory using joblib or pickle.
* **Success Metric:** The script runs and saves several .pkl files representing your trained MoE model.

**Module 7 & 8: The "Smart" Predictor & NAS**

* **Context for LLM Agent:** This is the upgrade. We will replace the simple formula-based predictor and the brute-force search loop with our trained AI. The new predictor will use the MoE model for more accurate estimations. The new search algorithm will be a "NAS-lite" evolutionary search that intelligently explores the configuration space to find the best solutions faster.
* **Task for LLM Agent:**
  1. **In core\_predictor.py**: Create a new function predict\_performance\_ai(model\_json, hardware\_config). This function will load the trained MoE models from Module 6 and use them to predict performance. It will have the same inputs and outputs as the "dumb" version.
  2. **In core\_search.py**: Create a new function run\_nas\_search(model\_json, constraints). This will be an evolutionary algorithm:
     + Start with a random population of 10 hardware configs.
     + In a loop for 20 generations:
       - Evaluate each config's "fitness" using the AI predictor (Module 7) and how well it meets constraints.
       - Select the top 5 "fittest" configs.
       - Create 5 new "offspring" configs by "mutating" (randomly changing a parameter) the best ones.
       - Repeat.
  3. **In app.py**: Simply change the button's function call from run\_simple\_search to run\_nas\_search.
* **Success Metric:** The app now uses the AI. The results should be more nuanced, and the search process (even if faked for speed in the UI) can be described as a "Next-Gen NAS Search."

**Phase 4: Final Polish & Demo Prep (Final Hour)**

**Goal:** Lock it down. Prepare for presentation.

* **Task:**
  1. **NO NEW FEATURES.**
  2. Make the UI look clean. Add explanations, titles, and team names.
  3. Pre-load a model in the app so the demo starts instantly.
  4. Write a 2-minute demo script.
  5. Record a video of the demo working perfectly. This is your backup if anything fails live.
  6. Prepare 3-4 slides: Problem, Our Solution (show a diagram of the E2E flow), Demo, Vision/Impact.
* **Success Metric:** You have a compelling, bug-free demo ready to present, and you're not coding 5 minutes before the deadline.