

# Scalable AI: Bridging Theory, Understanding, and Practice

Spring 2026

## Assignment 4: Applications

Release date: April 5th

Due date: April 23

Assignments 1–3 focused on the model lifecycle in progressively richer settings: parallelism, pre-training, and post-training. In this assignment, the emphasis shifts from *training a model* to *turning a model into a useful system*. Your job is to build an end-to-end AI application around an allowed model, choose and justify a serving stack, build the application pipeline that makes it usable, design benchmarks that measure both quality and systems performance, and present a working application.

A strong submission treats the application as a **full-stack artifact**: *model choice + prompting + application logic + serving + evaluation + demo*. A bare notebook that calls a model once is not enough.

### Overview

You will design, implement, benchmark, and present an AI application. Your application may be a coding assistant, a grounded question-answering system, a document or data assistant, a research or workflow copilot, a tutoring system, or something more ambitious. The application must be **end-to-end and interactive**: it should expose a usable interface or API, incorporate application-specific logic beyond raw prompting, and be evaluated under a workload that resembles how the application would actually be used.

#### What this assignment is about

This is an **applications** assignment, not a retraining assignment. You may use models from earlier assignments or one of the allowed NVIDIA checkpoints (`nvidia/NVIDIA-Nemotron-3-Nano-30B-A3B-BF16` or `nvidia/NVIDIA-Nemotron-3-Super-120B-A12B-BF16`), but the majority of your effort should go into:

- designing a useful application,
- choosing and justifying an inference backend,
- building the application pipeline,
- benchmarking and prompt tuning, and
- demonstrating a working system.

### Allowed starting points

You may build your application on top of one of the following:

1. a checkpoint produced by your team in earlier assignments (including your own base model or a derivative built from it), or
2. one of the allowed NVIDIA post-trained checkpoints: `nvidia/NVIDIA-Nemotron-3-Nano-30B-A3B-BF16` or `nvidia/NVIDIA-Nemotron-3-Super-120B-A12B-BF16`.

In all cases, you must clearly state **exactly which checkpoint is deployed in the final application**. If you use a post-trained derivative, you should say what the base checkpoint was and what additional adaptation was applied.

**Important.** You may not use a proprietary hosted LLM as the core model behind the application. External utilities such as vector databases, parsers, rerankers, search systems, or embedding models are allowed if you disclose them clearly.

## Learning objectives

After completing this assignment, you should be able to:

1. formulate a realistic AI application around a model and target user workflow,
2. choose an inference stack based on measured evidence rather than convenience or popularity,
3. build an application pipeline that makes the underlying model useful for end users,
4. design benchmarks for both **quality** and **systems performance**,
5. tune prompts or pipeline components using a development set without overfitting the final test set, and
6. communicate the tradeoffs among quality, latency, throughput, memory, and engineering complexity.

## Examples of acceptable application directions

The examples below are suggestions only. You are encouraged to propose something else if it is meaningful, measurable, and buildable within the assignment timeline.

Application family	Example ideas
Coding	repository-aware code assistant, bug-fix assistant, test generation, patch suggestion, CLI copilot
Grounded QA / RAG	course-notes assistant, paper assistant, enterprise-document QA, retrieval-grounded help desk
Agentic workflows	issue triage, notebook copilot, data-analysis assistant, research assistant with tools
Structured outputs	SQL assistant, extraction system, planning assistant, tutoring or grading assistant
Advanced / fancy	multi-agent system, tool-using coding environment, hybrid retrieval + planning pipeline, specialized domain copilot

## What you must do

Your submission must satisfy all four tasks below.

## 1 Task 0: Design an application (20 points)

Design a concrete application around an allowed model. The application should be useful to a plausible user and should require more than a single raw prompt to the model.

At minimum, you must specify:

1. **Target user and use case.** Who is the application for, and what job is it helping them do?
2. **Interaction design.** What does the user provide, what does the system produce, and what does one end-to-end interaction look like?
3. **Success criteria.** What does a successful outcome mean for this application?
4. **Architecture.** Include a system diagram showing the model, serving layer, application logic, and any retrieval/tools/storage used by the application.
5. **Failure modes.** Identify at least three likely failure modes and how your design attempts to mitigate or detect them.

### Minimum bar for Task 0

A chatbot wrapper around a model with no application-specific logic will receive limited credit. Your design should include at least one meaningful application layer, such as retrieval, tool use, routing, output validation, structured generation, state handling, safety checks, or another nontrivial system component.

## 2 Task 1: Choose an inference framework and build the application pipeline (25 points)

You may use any reasonable inference backend: **TensorRT-LLM**, **vLLM**, **SGLang**, **Dynamo**, **TGI**, **llama.cpp**, or another framework of your choice. However, your choice must be **numerically justified**. It is not enough to say that a framework is popular, easy to use, or recommended online.

### 1A. Serving-stack choice

You must compare either:

- **two candidate backends**, or
- **two substantially different deployment configurations** if only one backend is feasible for your model or hardware.

Examples of meaningful configuration changes include batching policy, quantization, tensor parallel degree, paged KV cache settings, streaming vs. non-streaming, speculative decoding, prefix caching, chunked prefill, or request scheduling policy.

Your comparison must keep the following fixed unless you explicitly justify a change:

- model checkpoint,
- hardware,
- prompt/workload mix,
- max input length and max output length, and
- measurement methodology.

## 1B. Application pipeline

You must build **substantial application logic** between the user-facing interface and the model server. A thin HTTP wrapper around a backend is not enough. The application pipeline is where most of your engineering effort should go—it is what turns a raw model into a system that is actually useful for the target user.

Examples of the kinds of components a strong pipeline might include:

- request validation and schema checking,
- prompt templating or context construction,
- retrieval and context injection,
- tool dispatch or tool calling,
- structured output validation,
- session or conversation state,
- caching,
- routing or fallback logic,
- safety or guardrail checks,
- logging, tracing, or metrics collection,
- rate limiting or admission control.

The above are examples, not a checklist. Your pipeline should reflect the complexity of your application—the more ambitious the application, the more interesting the pipeline.

**What numerical justification looks like.** A good justification sounds like: “We chose Backend X because at our target workload (2k-token context, 256-token outputs, concurrency 8), it achieved  $1.6\times$  higher throughput than Backend Y while keeping p95 latency within our interactive budget and preserving application quality on the held-out set.” A bad justification sounds like: “We used vLLM because everyone uses it.”

## 3 Task 2: Design benchmarks, measure quality, and prompt tune (30 points)

You must design an evaluation plan that measures both **application quality** and **systems performance**. You should use your benchmarks both to guide design choices and to justify your final system.

### 2A. Quality benchmark

Create a benchmark appropriate for your application.

Requirements:

1. Build a **development set** used for prompt or pipeline tuning and a separate **held-out test set** used only for final evaluation.
2. Use at least one **application-appropriate quality metric**. Examples include:
  - coding: pass@1, unit-test pass rate, compile success, repair success,
  - QA / RAG: exact match, F1, answer faithfulness, citation correctness, abstention accuracy,

- agentic workflows: task success rate, tool-call accuracy, steps to completion, intervention rate,
  - structured outputs: execution accuracy, schema validity, exact match, field-level F1.
3. If automatic metrics are insufficient, add a **human evaluation rubric** and report sample size, rubric definition, and inter-rater procedure if applicable.
  4. Report at least **five representative successes/failures** and analyze what the system gets right or wrong.

## 2B. Prompt tuning / pipeline tuning

Use the development set to tune the application. Tuning can target prompts, system instructions, retrieval strategy, reranking, tool descriptions, output format, decoding settings, or routing logic.

Requirements:

- Compare at least **three prompt or pipeline variants** on the development set.
- Select a final variant using the development set only.
- Run the final chosen variant **once** on the held-out test set and report the result separately.

## 2C. Systems benchmark

Measure how the deployed pipeline behaves as a served application.

At minimum, report:

- p50 and p95 end-to-end latency,
- throughput (requests/s, tokens/s, or task completions/s as appropriate),
- time-to-first-token (for streaming applications) or equivalent startup latency,
- peak memory usage or GPU memory footprint,
- at least one workload sweep over **concurrency**, and
- at least one workload sweep over **input length**, **output length**, or both.

All performance results must state:

- GPU type and count,
- precision / quantization,
- backend and version,
- maximum context length,
- maximum generation length,
- concurrency or batch configuration,
- whether streaming is enabled, and
- the exact request mix used in the benchmark.

### Very important

Your evaluation must measure the **actual deployed application pipeline**, not only offline single-example generations in a notebook. If retrieval, tools, routing, or validation are part of the final application, then at least one of your main benchmark tables must measure that full

end-to-end system.

#### 4 Task 3: Build and present the application (25 points)

Build the final application and present it.

Your final system must include:

1. a working user-facing interface or API (web UI, CLI, notebook app, or service endpoint),
2. the final served model configuration,
3. the application pipeline components,
4. a reproducible benchmark script or evaluation harness, and
5. a short presentation and live demo at checkoff (with a backup video if your live demo is brittle).

During the presentation/checkoff, you should be prepared to explain:

- why the application matters,
- why you chose the final backend,
- what the main quality and systems tradeoffs were,
- what failed and what you would improve next.

#### Minimum experimental expectations

To keep results meaningful and comparable, every submission must include the following:

- a **held-out quality evaluation** for the final application,
- a **backend comparison** or equivalent deployment comparison,
- a **throughput-latency tradeoff plot** or table,
- a **quality comparison** across at least three prompt/pipeline variants on the development set,
- a **failure analysis** with representative examples, and
- a **demo** of the working application.

#### What to include in the report

Submit a concise report of at most 6 pages, excluding references and appendix. The report should contain:

1. **Application overview** and target user,
2. **system architecture diagram**,
3. **model/checkpoint description**,
4. **serving backend comparison and justification**,
5. **application pipeline description**,
6. **quality benchmark design**,
7. **systems benchmark design**,
8. **prompt/pipeline tuning results**,
9. **final end-to-end results**,

## 10. failure analysis and limitations.

The report must include at least:

- one architecture figure,
- one backend-comparison table,
- one quality-results table,
- one latency/throughput figure or table, and
- one qualitative failure-analysis figure or table.

## Deliverables

Submit the following:

1. **Report PDF** (max 6 pages, excluding references and appendix),
2. **Code repository** or code archive with instructions to run the application and benchmarks,
3. **README** describing dependencies, hardware, launch instructions, and how to reproduce the main results,
4. **Benchmark artifacts** (tables, plots, or raw CSV/JSON logs),
5. **Slides PDF** for the short presentation, and
6. **Demo artifact** (live demo at checkoff; backup video recommended).

If your application depends on external data, APIs, or services, disclose them clearly in the README and report. If a TA cannot understand what is running where, reproducibility credit will be reduced.

## Grading rubric

Category	Points	What we are looking for
Task 0: Application design	20	Clear use case, meaningful application logic, sensible success criteria, architecture, and thoughtful failure modes.
Task 1: Serving stack + application pipeline	25	A justified backend choice based on measured evidence, plus real application logic rather than a thin wrapper.
Task 2: Benchmarks + prompt tuning	30	Strong quality benchmark, strong systems benchmark, dev/test separation, prompt or pipeline ablations, and careful analysis.
Task 3: Final build + presentation	25	A working application, convincing demo, reproducible setup, and clear presentation of trade-offs and limitations.
<b>Total</b>	<b>100</b>	

## Submission notes

- The course late policy applies.
- Be honest about what parts of the system are robust and what parts are still brittle.
- If something only works under very specific settings, document that clearly.
- If you use any third-party components, cite or acknowledge them appropriately.

### Checklist before you submit

Before submitting, make sure you can answer “yes” to all of the following:

- Did we build a real application rather than just call a model from a notebook?
- Did we choose the serving backend based on *measured* evidence?
- Did we build application logic that makes the system usable?
- Did we evaluate both *quality* and *throughput/latency/etc.*?
- Did we tune prompts/pipeline on a dev set and keep the test set held out?
- Did we benchmark the *deployed pipeline*, not just offline generations?
- Can we demo the application and explain its tradeoffs clearly?

*Build something useful. Measure it carefully. Justify your choices. Show us the system.*