1.3 LIGHTWEIGHT EVALUATION & SAFETY

As LLMs become central to your applications, you need ways to measure their performance and ensure they behave safely. This module covers practical evaluation strategies and responsible AI considerations.

LEARNING OBJECTIVES

By the end of this module, you will:

- Understand why evaluation matters for LLM applications
- Know basic evaluation signals for assessing LLM outputs
- Be aware of safety and responsible use considerations
- Understand how evaluation integrates with the accelerator

WHY EVALUATION MATTERS

THE PROBLEM

LLMs are powerful but unpredictable. Without evaluation, you risk:

Hallucinations: Model generates plausible but false information

```
Q: "What is IBM's revenue in 2025?"
Bad Answer: "IBM's revenue in 2025 was $87.4 billion." [Made up number]
```

Inconsistent Answers: Same question, different responses

```
Monday: "The capital of Australia is Sydney."
Tuesday: "The capital of Australia is Canberra." [Correct]
```

Business Impact: Wrong answers can lead to: - Lost customer trust - Compliance violations - Financial losses - Reputational damage

THE SOLUTION

Systematic evaluation helps you: - Catch problems before they reach users - Compare different models or prompts - Track quality over time - Build confidence in your system

SIMPLE EVALUATION SIGNALS

1. CORRECTNESS (GROUND TRUTH COMPARISON)

What it is: Does the answer match known correct information?

How to measure:

```python
def evaluate_correctness(model_answer: str, ground_truth: str) -> int:
    """Returns 1 if correct, 0 if incorrect"""
    # Simple exact match
    if model_answer.strip().lower() == ground_truth.strip().lower():
        return 1

    # Or use semantic similarity
    similarity = compute_similarity(model_answer, ground_truth)
    return 1 if similarity > 0.8 else 0
```

Example:

```
Question: "What year was IBM founded?"
Model Answer: "1911"
Ground Truth: "1911"
Score: 1 (Correct)

---

Question: "What year was IBM founded?"
Model Answer: "1920"
Ground Truth: "1911"
Score: 0 (Incorrect)
```

When to use: When you have known correct answers (test sets, benchmarks)

SAFETY & RESPONSIBLE USE

POTENTIAL RISKY CATEGORIES

LLMs can potentially generate harmful content in these categories:

1. **Personal Information (PII)**
   - Social security numbers, credit cards, passwords
   - Addresses, phone numbers, email addresses
2. **Harmful Content**
   - Hate speech, discrimination, harassment
   - Violence, self-harm, dangerous activities
   - Illegal activities, fraud schemes
3. **Misinformation**
   - Medical advice without disclaimers
   - Financial advice as fact
   - False claims about public figures
4. **Bias and Fairness**
   - Stereotyping based on protected attributes
   - Unfair treatment of groups
   - Lack of representation
5. **Privacy Violati**

HOW THIS TIES INTO THE ACCELERATOR

EVALUATION ENTRY POINTS

1. `tools/eval_small.py`

Purpose: Run a small evaluation dataset through your RAG system

What you’ll implement on Day 2-3:

```
# tools/eval_small.py

import pandas as pd
from accelerator.rag.pipeline import RAGPipeline

def evaluate_rag_system(test_file: str, output_file: str):
    """
    Evaluate RAG system on a test set

    Args:
        test_file: CSV with columns [question, ground_truth, category]
        output_file: Where to save results
    """
    # Load test data
    df = pd.read_csv(test_file)

    # Initialize pipeline
    pipeline = RAGPipeline()

    results = []
    for idx, row in df.iterrows():
        # Generate answer
        response = pipeline.answer_question(row['question'])
```

2. `accelerator/assets/notebook/Analyze_Log_and_Feedback.ipynb`

Purpose: Analyze logs and user feedback from the production service

What it does: - Load logs from `service/api.py` - Analyze patterns: - Most common questions - Average

HOW THIS CONNECTS TO LAB 1.3

WHAT YOU'LL BUILD

In Lab 1.3, you'll create a **micro-evaluation framework**:

1. **Test set**: 5-10 diverse prompts
2. **Data collection**: Run prompts through Ollama and watsonx
3. **Rating rubric**: Apply manual ratings (correctness, clarity, style)
4. **Analysis**: Compare backends, identify patterns

EXAMPLE OUTPUT (DATAFRAME)

prompt	backend	answer	correctness	clarity	style_match
"Summarize AI…"	ollama	"AI is…"	4	5	4
"Summarize AI…"	watsonx	"Artificial…"	5	5	5
"Extract emails…"	ollama	"john@…"	5	4	5

SKILLS YOU'LL DEVELOP

- Programmatic evaluation loops
- Rating rubric design
- Comparative analysis

REFERENCE NOTEBOOKS

GOVERNANCE & EVALUATION

`ibm-watsonx-governance-evaluation-studio-getting-started.ipynb`: - Shows watsonx.governance evaluation features - Demonstrates automated evaluation at scale - Connects to compliance tracking

From this, you’ll learn: - How to structure evaluation datasets - Metrics that matter for enterprise applications - Integration with governance workflows

Progression:

```
Lab 1.3 (Manual, 10 questions)
    ↓
eval_small.py (Automated, 100 questions)
    ↓
Evaluation Studio (Continuous, production scale)
```

BEST PRACTICES FOR EVALUATION

✅ DO

1. **Start simple:** Don’t over-engineer evaluation initially
2. **Automate what you can**: Manual review doesn’t scale
3. **Track over time**: Evaluation is ongoing, not one-time
4. **Test edge cases**: Don’t just test happy paths
5. **Involve stakeholders**: Domain experts should validate quality
6. **Version everything:** Track prompts, models, and test sets

❌ DON’T

1. **Skip evaluation**: “It looks good” isn’t enough
2. **Rely solely on accuracy**: Context matters (latency, safety, cost)
3. **Forget about drift**: Models and data change over time
4. **Ignore user feedback**: Real usage reveals issues testing doesn’t
5. **Over-optimize for metrics**: Gaming metrics != real quality

EVALUATION MATURITY MODEL

LEVEL 1: AD HOC

- Manual testing with a few examples
- “Looks good to me” approval
- No systematic tracking

LEVEL 2: BASIC (← LAB 1.3 TARGETS THIS)

- Small test set (10-50 examples)
- Manual rubric
- Occasional re-evaluation

LEVEL 3: SYSTEMATIC (← `eval_small.py` TARGETS THIS)

- Curated test set (100-500 examples)
- Automated metrics where possible
- Regular evaluation runs
- Version control for prompts and results

LEVEL 4: CONTINUOUS (← PRODUCTION GOAL)

- Large test set + production monitoring
- Automated evaluation pipeline

KEY TAKEAWAYS

- **Evaluation is essential**: Don’t deploy LLMs without systematic quality checks
- **Start simple**: Basic metrics beat no metrics
- **Safety first**: Proactively mitigate risks with guardrails
- **Iterate**: Evaluation frameworks evolve with your application
- **Automate**: Scale evaluation as your system scales

Next: Let’s build your first evaluation framework in Lab 1.3!

1.0 LLM CONCEPTS & ARCHITECTURE

Welcome to Day 1 of the watsonx Workshop! Today we'll explore Large Language Models (LLMs), their architecture, and how to work with them effectively.

LEARNING OBJECTIVES

By the end of this module, you will:

- Understand core LLM terminology and constraints
- Compare local vs managed LLM deployment models
- Know how LLMs fit into production architectures
- Understand key parameters that control model behavior

WHAT IS A LARGE LANGUAGE MODEL?

A **Large Language Model (LLM)** is a neural network trained on vast amounts of text data to understand and generate human-like text. Think of it as a powerful pattern-matching engine that has learned the statistical relationships between words, phrases, and concepts.

KEY CHARACTERISTICS

Scale: LLMs contain billions of parameters (weights) that encode knowledge learned from training data. For example: - GPT-3: 175 billion parameters - Llama 3.2: 1-3 billion parameters (smaller variants) - Granite 13B: 13 billion parameters

Training Data: Models are trained on diverse text sources including: - Books, articles, and documentation - Web pages and forums - Code repositories - Scientific papers

Capabilities: Modern LLMs can: - Answer questions - Summarize documents - Write code - Translate languages - Extract structured information - Reason through problems (with varying degrees of success)

KEY CONCEPTS

TOKENS & TOKENIZATION

LLMs don’t work with words—they work with **tokens**. A token is a sub-unit of text that the model processes.

Examples: - “Hello” → 1 token - “watsonx.ai” → might be 2-3 tokens (depends on the tokenizer) - “AI” → 1 token - A space or punctuation can be its own token

Why this matters: - Models have **token limits** (context windows) - API costs are often calculated per token - Long documents need to be chunked to fit within token limits

Rule of thumb: - English: ~4 characters per token on average - Code: Often more tokens per line than natural language

CONTEXT WINDOW AND TRUNCATION

The **context window** is the maximum number of tokens a model can process at once. This includes both: - Your input (prompt) - The model’s output (completion)

Common context window sizes: - Llama 3.2 (1B): 128K tokens - Granite 13B: 8K tokens (some variants) - GPT-4: 8K-32K tokens (depending on version)

Truncation: If your input exceeds the context window, it gets truncated (cut off), which can lead to: - Missing important context - Incomplete responses - Errors

LOCAL VS MANAGED LLMS

You have two main deployment options for LLMs in production:

LOCAL LLMS (E.G., OLLAMA)

What it is: Running models on your own infrastructure (laptop, on-prem servers, private cloud).

Pros: - ✅ **Privacy:** Data never leaves your environment - ✅ **Control:** Full control over model versions and updates - ✅ **Cost:** No per-token API charges after initial setup - ✅ **Customization:** Can fine-tune models for specific use cases - ✅ **Offline:** Works without internet connectivity

Cons: - ❌ **Hardware requirements:** Need GPUs for acceptable performance - ❌ **Maintenance:** You manage infrastructure, updates, scaling - ❌ **Limited scale:** Constrained by your hardware resources - ❌ **Model selection:** Limited to models that fit in your hardware

Best for: - Prototyping and development - Privacy-sensitive applications - Organizations with existing GPU infrastructure - Small to medium workloads

Example tools: - **Ollama:** Easy local LLM management - **LM Studio:** GUI for local models - **vLLM:** High-performance inference server

MANAGED LLMS (E.G., WATSONX.AI)

What it is: Using LLMs via cloud APIs where the provider handles infrastructure.

COST & RESOURCE CONSIDERATIONS

GPU VS CPU

GPU (Graphics Processing Unit): - Designed for parallel computations - Essential for training LLMs - Greatly accelerates inference (10-100x faster than CPU) - **Cost**: $1,000 - $10,000+ per card (consumer to enterprise)

CPU (Central Processing Unit): - General-purpose computing - Can run small models (1-3B parameters) acceptably - Struggles with larger models (13B+) - **Cost**: Cheaper, already available in most systems

Memory requirements: - Rough estimate: Model needs ~2 bytes per parameter (for FP16) - Example: 13B model needs ~26 GB GPU memory

CLOUD COST DIMENSIONS

When using managed services like watsonx.ai:

Token-based pricing: - Input tokens: Text you send - Output tokens: Text generated - Typically: $0.0001 - $0.001 per token (varies by model)

Example calculation:

```
Prompt: 1,000 tokens
Response: 500 tokens
Cost: (1000 + 500) × $0.0002 = $0.30 per request
```

Cost optimization strategies: - Use smaller models when appropriate - Cache common responses -

WHERE THE ACCELERATOR FITS ARCHITECTURALLY

Throughout this workshop, we’ll reference the **RAG Accelerator**—a production-ready skeleton for building LLM applications. Here’s how it’s structured:

CORE ARCHITECTURE

```
accelerator/
├── rag/                    # RAG core logic
│   ├── pipeline.py        # Orchestrates retrieval + LLM
│   ├── retriever.py       # Vector DB queries (Elasticsearch/Chroma)
│   ├── prompt.py          # Shared prompt templates
│   └── embedder.py        # Text embedding logic
├── service/               # Production API
│   ├── api.py            # FastAPI microservice (POST /ask)
│   ├── deps.py           # Configuration & dependencies
│   └── models.py         # Request/response schemas
├── tools/                 # CLI utilities
│   ├── chunk.py          # Document chunking
│   ├── extract.py        # Text extraction from PDFs, docs
│   ├── embed_index.py    # Embedding and indexing pipeline
│   └── eval_small.py     # Evaluation harness
```

HOW LLMS FIT IN

On Day 1, we’re focusing on **pure LLM behavior** (no retrieval). This maps to:

Current state (Day 1):

```
# pipeline.py (simplified)
def answer_question(question: str) -> str:
    # Direct LLM call
```

REFERENCE NOTEBOOKS

The workshop includes several reference notebooks that show LLMs in production contexts:

RAG EXAMPLES (`labs-src/`)

- `use-watsonx-elasticsearch-and-langchain-to-answer-questions-rag.ipynb`
  - Full RAG pipeline with Elasticsearch
  - Shows prompt structure with context
- `use-watsonx-chroma-and-langchain-to-answer-questions-rag.ipynb`
  - Alternative vector DB (Chroma)
  - LangChain integration patterns

ACCELERATOR NOTEBOOKS (`accelerator/assets/notebook/`)

- `QnA_with_RAG.ipynb`
  - End-to-end Q&A with retrieval
  - Prompt engineering for RAG
- `Create_and_Deploy_QnA_AI_Service.ipynb`
  - Deploy RAG service to production
  - API endpoint creation

How to use these: - Don’t run them line-by-line on Day 1 - **Do** open them to see: - How prompts are structured - How LLM calls are instrumented - How outputs are validated

HOW THIS CONNECTS TO THE LABS

DAY 1 LABS (TODAY)

- **Lab 1.1**: Quick start with both Ollama and watsonx
  - Focus: Basic LLM calls, parameter tuning
  - No retrieval, just prompts → responses
- **Lab 1.2**: Prompt templates
  - Build reusable prompt patterns
  - Compare behavior across backends
- **Lab 1.3**: Micro-evaluation
  - Rate LLM outputs
  - Build a simple evaluation framework

DAY 2-3 LABS (UPCOMING)

- Add retrieval (RAG)
- Integrate with the accelerator
- Build production-ready pipelines
- Add orchestration and agents

Mental model: - **Day 1** = Understanding the LLM building block - **Day 2** = LLM + retrieval (RAG) - **Day 3** = LLM + retrieval + orchestration (agents)

FURTHER READING

OFFICIAL DOCUMENTATION

- IBM Granite Models
- watsonx.ai Documentation
- Ollama Documentation

PROMPT ENGINEERING

- OpenAI Prompt Engineering Guide
- Anthropic Prompt Engineering
- Granite Prompting Guide

LLM CONCEPTS

- Hugging Face NLP Course
- LLM Training & Inference
- Understanding Tokenization

RESPONSIBLE AI

- IBM AI Ethics
- Guardrails for LLMs

SUMMARY

You now understand:

- ✅ What LLMs are and how they work at a high level
- ✅ Key concepts: tokens, context windows, temperature, top-k/top-p
- ✅ Trade-offs between local and managed deployments
- ✅ Cost considerations for LLM applications
- ✅ How LLMs fit into the accelerator architecture

Next: Let’s get hands-on with Lab 1.1 and actually run some prompts!

1.2 PROMPT PATTERNS & TEMPLATES

Understanding how to structure prompts effectively is crucial for getting reliable, high-quality outputs from LLMs. This module covers common prompt patterns and how to build reusable templates.

LEARNING OBJECTIVES

By the end of this module, you will:

- Recognize common prompt patterns and when to use them
- Understand why structure matters in prompt engineering
- Know how to create reusable prompt templates
- See how the accelerator uses prompts in production

CORE PROMPT PATTERNS

1. INSTRUCTION PROMPTS

The simplest pattern: give the model a clear instruction.

Structure:

```
[Instruction]
```

Examples:

```
Summarize this text in 3 sentences.

Extract all email addresses from the following document.

Translate this paragraph to French.
```

Best for: - Simple, well-defined tasks - When the model already knows what to do - Single-step operations

Tips: - Be specific and direct - Use action verbs (summarize, extract, translate, list) - Specify output format if needed

2. FEW-SHOT EXAMPLES

Provide examples of the task before asking the model to do it.

Structure:

```
[Instruction]
```

PROMPT DESIGN PRINCIPLES

1. CLARITY AND SPECIFICITY

Bad:

```
Tell me about AI.
```

Good:

```
Explain the difference between supervised and unsupervised machine learning
in 3 paragraphs, with one example of each.
```

Why it matters: Vague prompts lead to vague, unfocused responses.

2. ROLE AND PERSONA

Give the model a role to frame its responses.

Structure:

```
You are a [role] with expertise in [domain].

[Task]
```

Examples:

```
You are a senior software architect with 15 years of experience in distributed systems.

Design a scalable architecture for a real-time chat application.
```

PROMPT TEMPLATES

WHAT IS A TEMPLATE?

A **prompt template** is a reusable pattern with placeholders for variable content.

Benefits: - **Consistency**: Same structure every time - **Maintainability**: Update once, apply everywhere - **Testability**: Easier to evaluate prompt changes - **Scalability**: Supports batch processing

SIMPLE PYTHON TEMPLATES

Using f-strings:

```
def summarize(text: str, length: int = 3) -> str:
    prompt = f"""Summarize the following text in {length} sentences:

{text}

Summary:"""
    return llm.generate(prompt)
```

Using str.format():

```
TEMPLATE = """You are a helpful assistant.

Task: {task}

Input: {input_text}
```

HOW THE ACCELERATOR USES PROMPTS

The accelerator centralizes prompt logic in `rag/prompt.py`:

CURRENT STRUCTURE

```
# accelerator/rag/prompt.py

SYSTEM = """You are a careful and accurate assistant.
You answer questions based on provided context.
If you cannot find the answer in the context, say so."""

USER_TEMPLATE = """Context:
{context}

Question: {question}

Answer:"""
```

ON DAY 2-3, YOU’LL EXTEND THIS

Multi-turn conversations:

```
CHAT_TEMPLATE = """You are a helpful assistant. Use the following context to answer questions.

Context:
{context}

Conversation history:
{history}
```

REFERENCE NOTEBOOKS

RAG PROMPT EXAMPLES

`use-watsonx-chroma-and-langchain-to-answer-questions-rag.ipynb`:

```
# Example from the notebook
from langchain.prompts import PromptTemplate

rag_prompt = PromptTemplate(
    template="""Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer.

{context}

Question: {question}
Helpful Answer:""",
    input_variables=["context", "question"],
)
```

`QnA_with_RAG.ipynb` (accelerator): - Shows how context is concatenated - Demonstrates citation patterns - Includes error handling for edge cases

KEY OBSERVATIONS

1. **Context injection**: Always happens before the question
2. **Structured formats**: JSON/XML for tool calling
3. **Safety prompts**: Explicitly state boundaries

EXAMPLES TO REUSE IN LAB 1.2

EXAMPLE 1: SUMMARIZATION

Template:

```
SUMMARIZE_TEMPLATE = """Summarize the following text in {num_sentences} sentences.
Focus on the main points and key takeaways.

Text:
{text}

Summary:"""
```

Usage:

```
prompt = SUMMARIZE_TEMPLATE.format(
    num_sentences=3,
    text="[Your long text here]"
)
```

EXAMPLE 2: STYLE TRANSFER

Template:

```
REWRITE_TEMPLATE = """Rewrite the following text in a {target_tone} tone:

Original text:
{original}
```

CONNECTION TO LABS

LAB 1.2: PROMPT TEMPLATES

In this lab, you'll:

1. **Create templates** for:
   - Summarization
   - Style rewrite
   - Q&A with context
2. **Implement in both environments**:
   - `prompt_patterns_ollama.ipynb` (local)
   - `prompt_patterns_watsonx.ipynb` (managed)
3. **Compare results**:
   - Same template, different models
   - Measure quality and consistency

LOOKING AHEAD

Day 2: These templates become the foundation for RAG prompts **Day 3**: Templates extended for multi-turn agents and tool calling

BEST PRACTICES SUMMARY

✅ DO

- Start with simple, clear instructions
- Use few-shot examples for structured outputs
- Specify output format explicitly
- Test prompts with edge cases
- Version control your templates
- Measure prompt performance systematically

❌ DON’T

- Use vague or ambiguous language
- Mix multiple tasks in one prompt
- Assume the model knows your context
- Forget to handle error cases
- Over-engineer prompts prematurely

KEY TAKEAWAYS

- **Prompts are code**: Treat them with the same rigor as application code
- **Structure matters**: Well-structured prompts are more reliable
- **Templates enable scale**: Reusable patterns save time and improve consistency
- **Test and iterate**: Prompt engineering is empirical—what works for one task may not work for another

Next: Time to build these patterns hands-on in Lab 1.2!

DAY 1 – LLMS & PROMPTING - COMPLETE WORKSHOP GUIDE

Date: Day 1 of watsonx Workshop

Duration: 8 hours (4 hours theory + 4 hours labs)

Track: Core/Granite

WORKSHOP SCHEDULE

MORNING SESSION (4 HOURS) - THEORY

Time	Duration	Topic	Description
9:00 - 9:15	15 min	Welcome & Setup Check	Verify Day 0 completion
9:15 - 10:30	75 min	**1.0 LLM Concepts**	Core concepts, local vs managed, architecture
10:30 - 10:45	15 min	Break	
10:45 - 11:45	60 min	**1.2 Prompt Patterns**	Patterns, templates, best practices
11:45 - 12:00	15 min	Q&A	
12:00 - 1:00	60 min	Lunch	

AFTERNOON SESSION (4 HOURS) - LABS

Time	Duration	Topic	Description
1:00 - 1:45	45 min	**Lab 1.1**	Quickstart in both environments

LEARNING PATH

```
Day 0 (Prerequisites)
    ↓
Day 1 Morning: Theory
    → 1.0 LLM Concepts
    → 1.2 Prompt Patterns
    ↓
Day 1 Afternoon: Labs
    → Lab 1.1: Quickstart
    → Lab 1.2: Templates
    → Lab 1.3: Evaluation
    ↓
Day 2: RAG (Retrieval-Augmented Generation)
```

MATERIALS PROVIDED

THEORY DOCUMENTS

1. `llm-concepts.md` - Core LLM concepts and architecture
2. `prompt-patterns-theory.md` - Prompt engineering patterns
3. `eval-safety-theory.md` - Evaluation and safety

LAB INSTRUCTIONS

1. `lab-1-quickstart-two-envs.md` - Lab 1.1 guide
2. `lab-2-prompt-templates.md` - Lab 1.2 guide
3. `lab-3-micro-eval.md` - Lab 1.3 guide

NOTEBOOKS (TO BE CREATED BY PARTICIPANTS)

1. `ollama_quickstart.ipynb` - Ollama experiments
2. `watsonx_quickstart.ipynb` - watsonx.ai experiments
3. `prompt_patterns_ollama.ipynb` - Ollama templates
4. `prompt_patterns_watsonx.ipynb` - watsonx templates
5. `micro_evaluation.ipynb` - Evaluation framework

REFERENCE MATERIALS

- `labs-src/` - Reference RAG notebooks

LEARNING OBJECTIVES BY MODULE

1.0 LLM CONCEPTS

- ✅ Understand tokens, context windows, parameters
- ✅ Compare local vs managed deployments
- ✅ Know cost and resource considerations
- ✅ Understand accelerator architecture

1.2 PROMPT PATTERNS

- ✅ Recognize common prompt patterns
- ✅ Build reusable templates
- ✅ Apply prompt engineering principles
- ✅ Design production prompts

LAB 1.1: QUICKSTART

- ✅ Run prompts in Ollama and watsonx
- ✅ Modify parameters (temperature, max_tokens)
- ✅ Compare outputs and latency
- ✅ Connect to accelerator

LAB 1.2: TEMPLATES

PREREQUISITES CHECKLIST

Before starting Day 1, ensure:

- ✅ Day 0 completed
- ✅ `simple-ollama-environment` working
- ✅ `simple-watsonx-enviroment` working with credentials
- ✅ Jupyter accessible in both environments
- ✅ Ollama has at least one model pulled (e.g., `qwen2.5:0.5b-instruct`)
- ✅ watsonx.ai credentials verified (API key, URL, project ID)
- ✅ `watsonx-workshop` repo cloned (for accelerator reference)

KEY CONCEPTS SUMMARY

TOKENS

- Sub-units of text that LLMs process
- ~4 characters per token (English average)
- Context window = max tokens (input + output)

TEMPERATURE

- 0.0 = Deterministic, focused
- 0.7-1.0 = Balanced
- 1.5+ = Creative, unpredictable

PROMPT PATTERNS

1. **Instruction**: Direct command
2. **Few-shot**: Examples before task
3. **Chain-of-thought**: Step-by-step reasoning
4. **Style transfer**: Rewrite in different tone
5. **Summarization**: Condense content

EVALUATION SIGNALS

1. **Correctness**: Matches ground truth?

INSTRUCTOR NOTES

MORNING SESSION TIPS

- **LLM Concepts**: Use diagrams for architecture
- **Prompt Patterns**: Live demo with watsonx Prompt Lab
- Keep theory interactive with questions
- Relate concepts to students' use cases

AFTERNOON SESSION TIPS

- **Lab 1.1**: Ensure all students complete before moving on
- **Lab 1.2**: Encourage creativity in template design
- **Lab 1.3**: Form small groups for evaluation discussions
- Circulate during labs to answer questions
- Have backup notebooks ready for students with issues

COMMON ISSUES

1. **Ollama not running**: Check Docker container or service
2. **watsonx 401 errors**: Verify credentials in `.env`
3. **Rate limits**: Remind students to pace requests
4. **Python environment**: Ensure correct kernel selected

SUCCESS CRITERIA

By end of Day 1, students should be able to:

1. **Explain** how LLMs work at a high level
2. **Compare** local and managed LLM deployments
3. **Write** effective prompts for different tasks
4. **Build** reusable prompt templates in Python
5. **Evaluate** LLM outputs systematically
6. **Run** notebooks in both Ollama and watsonx environments
7. **Understand** how LLMs fit into the accelerator architecture

HOMEWORK (OPTIONAL)

1. **Expand test set**: Add 10 more diverse prompts to Lab 1.3
2. **Try different models**:
   - Ollama: `llama3.2:3b`, `qwen2.5:1.5b`
   - watsonx: Try different Granite variants
3. **Advanced prompting**: Implement a multi-turn conversation pattern
4. **Read ahead**: Review Day 2 materials on RAG concepts

CONNECTIONS TO FUTURE DAYS

DAY 2 (RAG)

- Today’s prompts → prompts with retrieved context
- Single LLM call → retrieval + LLM pipeline
- Manual evaluation → automated RAG metrics (retrieval quality, answer quality)

DAY 3 (AGENTS & ORCHESTRATION)

- Static prompts → dynamic tool-calling prompts
- Single-turn → multi-turn conversations
- Basic evaluation → production monitoring

RESOURCES

DOCUMENTATION

- IBM Granite Models
- watsonx.ai Docs
- Ollama Docs

PROMPT ENGINEERING

- OpenAI Prompt Guide
- Granite Prompting Guide

COMMUNITY

- IBM Granite GitHub
- watsonx Community

FEEDBACK & QUESTIONS

DURING WORKSHOP

- Use chat/Slack for quick questions
- Raise hand for blocking issues
- Share interesting findings with the group

AFTER WORKSHOP

- Complete feedback survey
- Share lab solutions with peers
- Join community discussions

DAY 1 COMPLETION CHECKLIST

Theory: - ✅ Attended 1.0 LLM Concepts session - ✅ Attended 1.2 Prompt Patterns session - ✅ Attended 1.3 Evaluation & Safety session

Labs: - ✅ Completed Lab 1.1 (Quickstart) - ✅ Completed Lab 1.2 (Templates) - ✅ Completed Lab 1.3 (Evaluation)

Deliverables: - ✅ Working notebooks in both environments - ✅ Prompt templates created - ✅ Evaluation results CSV generated

Understanding: - ✅ Can explain LLM concepts - ✅ Can write effective prompts - ✅ Can evaluate LLM outputs - ✅ Ready for Day 2 (RAG)

NEXT: DAY 2 PREVIEW

Tomorrow we’ll: 1. Add **retrieval** to our LLM calls (RAG) 2. Integrate with the **accelerator** codebase 3. Build a **production-ready** RAG service 4. Learn about **vector databases** and **embeddings**

Prepare by: - Reviewing today’s materials - Ensuring accelerator code is accessible - Thinking about documents you’d like to use for RAG

Congratulations on completing Day 1! 🎉

You’ve built a strong foundation in LLM fundamentals and prompt engineering. Tomorrow, we’ll take it to the next level with RAG.