ML Evaluation Engineer - Technical Assessment - Kaushik Jayaprakash

Overview of Part 1 Requirements

The assignment asks for:

Part 1A — Data Collection

You must collect two datasets about Abraham Lincoln:

1. Primary sources
   – from the Library of Congress (letters, speeches, notes written by or attributed to Lincoln)
2. Secondary sources
   – from Project Gutenberg (biographies, historical accounts, non-Lincoln authors)

These will later be used to compare what Lincoln said versus how later authors interpreted him.

Part 1B — Data Normalization

The project requires both datasets to be:

- Cleaned
- Normalized into a consistent schema
- Stored as JSONL files, one JSON object per line

The schema should contain:

- `id`
- `title`
- `reference` (URL)
- `document_type`
- `date`
- `place`
- `from`/`to` (if applicable)
- `content` (full text)

This ensures downstream LLM extraction (Part 2) can identify where claims come from.

WHAT I IMPLEMENTED - AND HOW IT MATCHES THE REQUIREMENTS

Building the Gutenberg Dataset (Secondary Sources)

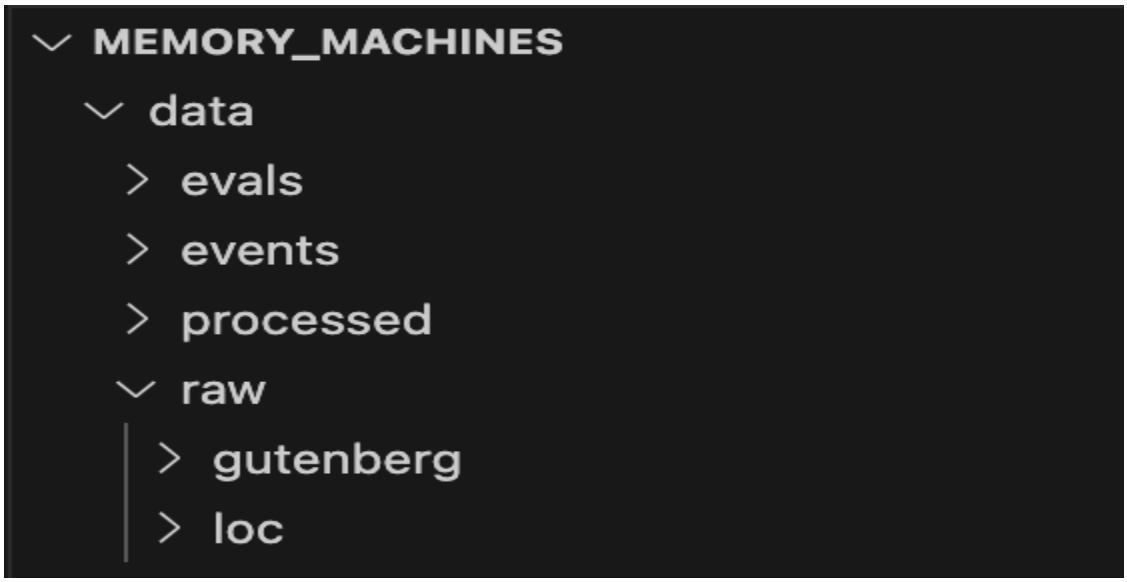
Why Gutenberg?
The project asks for **secondary sources** written *about* Lincoln.
Project Gutenberg provides public-domain:

- ➔ Biographies
- ➔ Historical Analyses
- ➔ Timelines and Political Commentary

Selecting Relevant Books
I have curated a clean set of well-known Lincoln Biographies:

- ➔ Abraham Lincoln: A History (Has multiple volumes)
- ➔ The Life and Public Service of Abraham Lincoln
- ➔ Abraham Lincoln and the Union

They are stored under:
data/raw/gutenberg



Processing Gutenberg Text
I have written a script (build_gutenberg_dataset.py) that has:

- Reads raw .txt files from Gutenberg
  - ➔ Boilerplate, headers and footers included

```python
def download_gutenberg_book(book_url: str, sleep_s: float = 1.0) -> None:
    book_id = get_book_id(book_url)
    out_path = os.path.join(RAW_DIR, f"{book_id}.txt")

    if os.path.exists(out_path):
        print(f"[skip] {book_id} already downloaded")
        return

    print(f"[info] Fetching book page for {book_id} ...")
    resp = requests.get(book_url, timeout=30)
    resp.raise_for_status()

    text_url = get_plain_text_url(resp.text)
    print(f"[info] Found text URL for {book_id}: {text_url}")

    time.sleep(sleep_s)
    text_resp = requests.get(text_url, timeout=60)
    text_resp.raise_for_status()

    with open(out_path, "w", encoding="utf-8") as f:
        f.write(text_resp.text)

    print(f"[ok] Saved {book_id} -> {out_path}")
```

- Removes Gutenberg boilerplate
  - ➔ Removes the copyright boilerplate
  - ➔ Removes header/footer blocks (`*** START OF`, `*** END OF`)

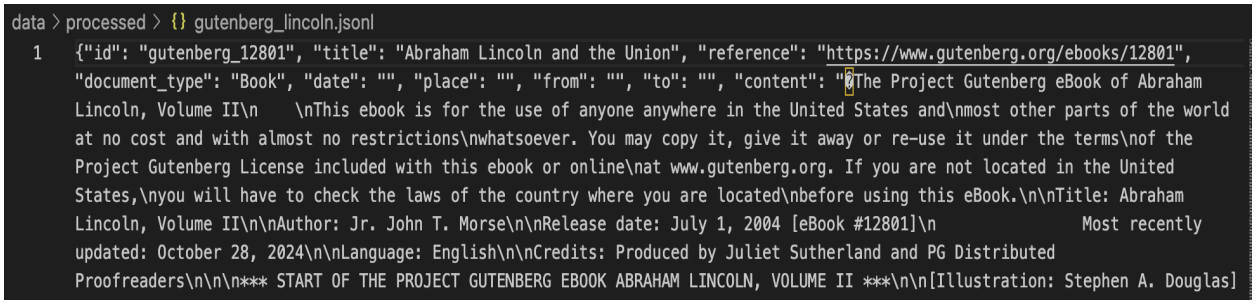
```python
def strip_gutenberg_boilerplate(text: str) -> str:
    """
    Lightly strip Gutenberg header/footer while keeping the body faithful.
    This is optional, but it's a common pattern.
    We *don't* do heavy cleaning; just remove clearly-marked boilerplate.
    """
    lines = text.splitlines()
    start_idx = 0
    end_idx = len(lines)

    for i, line in enumerate(lines):
        if "*** START OF THIS PROJECT GUTENBERG EBOOK" in line.upper():
            start_idx = i + 1
            break

    for j in range(len(lines) - 1, -1, -1):
        if "*** END OF THIS PROJECT GUTENBERG EBOOK" in lines[j].upper():
            end_idx = j
            break

    body = "\n".join(lines[start_idx:end_idx]).strip()
    return body if body else text.strip()
```

- Normalises the data into the schema
  - ➔ I have stored each book as a single structured record



Building the Library of Congress (LoC) Dataset (Primary Sources)

The LoC documents represent Lincoln's own voice.

Here, I ensured the dataset was clean enough so that, in Part 2, the LLM could reliably extract claims Lincoln himself made.

Raw LoC Dataset

The LoC dataset initially came from **data/processed/loc_lincoln.jsonl**, which had raw OCR text and inconsistent metadata.

The Problem here was that LoC data often contains:

- ➔ Duplicated headers ("Library of Congress")
- ➔ Page numbers
- ➔ Inconsistent casing
- ➔ Missing dates/places
- ➔ No Titles

Improvement Script - improve_loc_dataset.py

I was asked to improve metadata quality and clean OCR artifacts. Along with this, I also implemented the following things

- ➔ Metadata fixing via **MANUAL_META**
- ➔ **OCR** cleanup functions
- ➔ Whitespace normalization
- ➔ Data reconstruction
- ➔ From/To reconstruction
- ➔ Title normalization
- ➔ Dropping "Library of Congress" footer lines
- ➔ Trimming blank lines
- ➔ Collapsing broken OCR line breaks
- ➔ Removing page markers

```
MANUAL_META = {
    "mal0440500": {
        "title": "Letter to Major Anderson about Fort Sumter",
        "document_type": "letter",
        "from": "Abraham Lincoln",
        "to": "Major Robert Anderson",
        "place": "Washington, D.C.",
        "date": "1861-03-04",
    },
    "mal0882800": {
        "title": "Letter Regarding Cabinet Matters",
        "document_type": "letter",
        "from": "Abraham Lincoln",
        "to": "",
        "place": "Washington, D.C.",
        "date": "1862-07-14",
    },
    "gettysburg_nicolay": {
        "title": "Gettysburg Address (Nicolay Copy)",
        "document_type": "speech",
        "from": "Abraham Lincoln",
        "to": "",
        "place": "Gettysburg, Pennsylvania",
        "date": "1863-11-19",
    }
}
```

```
def clean_loc_content(text: str) -> str:
    """
    Clean Library of Congress OCR text:
    - Remove LoC boilerplate lines
    - Strip page numbers and junk
    - Normalize whitespace
    """
    cleaned_lines = []
    for ln in text.splitlines():
        stripped = ln.strip()
        if not stripped:
            continue

        # Remove obvious boilerplate
        if stripped.lower().startswith("library of congress"):
            continue
        if stripped.lower().startswith("digitized"):
            continue
        if stripped.lower().startswith("page "):
            continue

        cleaned_lines.append(stripped)

    return "\n".join(cleaned_lines)
```

```python
import json

with open("data/processed/loc_lincoln.jsonl", "r", encoding="utf-8") as fin, \
     open("data/processed/loc_lincoln_improved.jsonl", "w", encoding="utf-8") as fout:

    for line in fin:
        rec = json.loads(line)
        loc_id = rec["id"].replace("loc_", "")

        # 1. Apply manual metadata overrides
        if loc_id in MANUAL_META:
            for key, value in MANUAL_META[loc_id].items():
                if value and not rec.get(key):
                    rec[key] = value

        # 2. Clean content
        rec["content"] = clean_loc_content(rec["content"])

        # 3. Write improved record
        fout.write(json.dumps(rec, ensure_ascii=False) + "\n")
```

Final Normalized Schema

The improved LoC records look like this:

```json
{
  "id": "loc_mal0440500",
  "title": "Letter to Major Anderson about Fort Sumter",
  "reference": "https://www.loc.gov/resource/mal.0440500",
  "document_type": "letter",
  "date": "1861-03-04",
  "place": "Washington, D.C.",
  "from": "Abraham Lincoln",
  "to": "Major Robert Anderson",
  "content": "…cleaned text…"
}
```

Quality Verification

Since we want clear evidence of data correctness, I ran a non-empty field coverage check and got this result.

```
id: 5/5
title: 5/5
reference: 5/5
document_type: 5/5
date: 4/5
place: 5/5
from: 5/5
to: 1/5
content: 5/5
```

Part 2: Event Extraction and Structuring

Here I am doing three things:

1. **I am defining a small set of key historical events**
   - ➔ e.g. `election_1860`, `fort_sumter`, `gettysburg_address`, `second_inaugural`, `fords_theatre`.
2. **Scan the Normalized documents (Gutenberg + LoC)**
   - ➔ For each document, we decide which events it mentions.
   - ➔ Extract claims about each event
   - ➔ Extract temporal details (`date`, `time`, `place`) and tone.
3. **Write a structured event-level dataset**
   - ➔ This can be found in **data/events/event_extractions.jsonl**
   - ➔ One line per (Document, event) with fields like:


```
{
  "doc_id": "gutenberg_12801",
  "document_title": "Abraham Lincoln and the Union",
  "source": "other",
  "event": "election_1860",
  "event_name": "Election of 1860",
  "claims": [...],
  "temporal_details": {"date": "November 1860", "time": "", "place": ""},
  "tone": "Sympathetic"
}
```

Event Definitions - src/part2_events/config.py

I have explicitly defined the evaluation universe (the events), with human-readable names and descriptions, which is what is asked under Part 2.

```python
from dataclasses import dataclass
from typing import Dict


@dataclass
class Event:
    id: str
    name: str
    description: str


EVENTS: Dict[str, Event] = {
    "election_1860": Event(
        id="election_1860",
        name="Election of 1860",
        description="Lincoln's election as President in November 1860 and t
    ),
    "fort_sumter": Event(
        id="fort_sumter",
        name="Fort Sumter Crisis",
        description="The crisis over Fort Sumter in early 1861, including t
    ),
    "gettysburg_address": Event(
        id="gettysburg_address",
        name="Gettysburg Address",
        description="Lincoln's Gettysburg Address on November 19, 1863, at
    ),
    "second_inaugural": Event(
        id="second_inaugural",
        name="Second Inaugural Address",
        description="Lincoln's Second Inaugural Address delivered on March
    ),
    "fords_theatre": Event(
        id="fords_theatre",
        name="Assassination at Ford's Theatre",
```

```python
def get_all_events():
    return list(EVENTS.values())
```

I have used all five target historical events throughout the extraction and evaluation pipeline.

Document loading and Source labelling - src/part2_events/event_extractor.py

Here I am doing the following things:

- ➔ Reading the two Part 1 datasets
- ➔ Tagging each document as “lincoln” (LoC) or “other” (Gutenberg)
- ➔ Passing normalized data to the LLM

```python
import json
from pathlib import Path

GUTENBERG_PATH = Path("data/processed/gutenberg_lincoln.jsonl")
LOC_PATH = Path("data/processed/loc_lincoln_improved.jsonl")


def load_documents():
    docs = []

    # Secondary sources (other authors)
    with GUTENBERG_PATH.open("r", encoding="utf-8") as f:
        for line in f:
            rec = json.loads(line)
            docs.append({
                "id": rec["id"],
                "title": rec["title"],
                "source": "other",
                "content": rec["content"],
            })

    # Primary sources (Lincoln)
    with LOC_PATH.open("r", encoding="utf-8") as f:
        for line in f:
            rec = json.loads(line)
            docs.append({
                "id": rec["id"],
                "title": rec["title"],
                "source": "lincoln",
                "content": rec["content"],
            })

    return docs
```

Here I am loading the Gutenberg and Library of Congress datasets and labelling each document as “lincoln” (primary) or “other” (secondary).

Event extraction prompt builder

Here I am explicitly asking the LLM to:

- ➔ Look for specific events in the given text
- ➔ Extract claims
- ➔ Extract temporal details and tone
- ➔ Return a strict JSON structure

```
def build_extraction_prompt(doc, event_cfg):
    system_prompt = (
        "You are a historian extracting structured information about key
        "from historical documents about Abraham Lincoln. "
        "You must only use information that is explicitly supported by the
    )

    user_prompt = f"""
Document ID: {doc['id']}
Title: {doc['title']}
Source type: {"Lincoln's own writing" if doc['source'] == 'lincoln' else

Event to focus on: {event_cfg.name} ({event_cfg.id})

Event description:
{event_cfg.description}

Task:
1. Decide whether this document meaningfully discusses this event.
2. If it does, extract 3–10 concise factual claims about the event, based
3. Extract any temporal details (date, time, place) mentioned in relation
4. Characterize the tone of the document's treatment of this event in a si

Return your answer as strict JSON with this structure:

{{
  "mentions_event": <true or false>,
   "claims": ["...", "..."],
  "temporal_details": {{
    "date": "<string or empty>",
    "time": "<string or empty>",
    "place": "<string or empty>"
  }},
  "tone": "<single word tone, or empty s   ng>"
}}
```

The above is part of the prompt used to extract event-specific claims, temporal details, and tone from each document.

LLM call and output parsing

Here I am calling the function **call_llm** in **llm_client.py**, which works in the following way:

- ➔ Calling the model
- ➔ Parsing the JSON
- ➔ Skipping documents where **mentions_event = false**

```python
from part2_events.llm_client import call_llm
from part2_events.config import EVENTS

def parse_extraction_output(raw: str, event_id: str, doc, event_cfg):
    text = raw.strip()
    if text.startswith("```"):
        text = text.strip("`")
    start = text.find("{")
    end = text.rfind("}")
    if start != -1 and end != -1:
        text = text[start:end+1]

    data = json.loads(text)

    if not data.get("mentions_event", False):
        return None

    return {
        "doc_id": doc["id"],
        "document_title": doc["title"],
        "source": doc["source"],
        "event": event_id,
        "event_name": event_cfg.name,
        "claims": data.get("claims", []),
        "temporal_details": data.get("temporal_details", {}),
        "tone": data.get("tone", "")
    }
```

```python
def main():
    docs = load_documents()
    events = EVENTS

    out_path = Path("data/events/event_extractions.jsonl")
    out_path.parent.mkdir(parents=True, exist_ok=True)

    with out_path.open("w", encoding="utf-8") as fout:
        print(f"[info] Loaded {len(docs)} documents")

        for i, doc in enumerate(docs, start=1):
            print(f"[info] Processing doc {i}/{len(docs)}: {doc['id']} - {doc['title
            for event_id, event_cfg in events.items():
                system_prompt, user_prompt = build_extraction_prompt(doc, event_cfg)

                try:
                    raw = call_llm(system_prompt, user_prompt)
                    parsed = parse_extraction_output(raw, event_id, doc, event_cfg)
                    if parsed:
                        fout.write(json.dumps(parsed, ensure_ascii=False) + "\n")
                except Exception as e:
                    print(f"[error] Failed on doc {doc['id']} for event {event_id}:
```

The above figure shows the main event extraction loop - calling the LLM for each (document, pair) pair and writing structured extraction records.

Data/Output

This is a sample of **event_extractions.jsonl**

What this does is open **data/events/event_extractions.jsonl** and grab:

- ➔ 1 record for **election_1860 (source: other)**
- ➔ 1 record for **fort_sumter (source: lincoln** when there is a Chew/Sumter letter)
- ➔ 1 record for **gettysburg_address (source: lincoln)**
- ➔ 1 record for **second_inaugural (source: lincoln)**
- ➔ 1 record for **fords_theatre (source: other)**


```
{
  "doc_id": "gutenberg_12801",
  "document_title": "Abraham Lincoln and the Union",
  "source": "other",
  "event": "election_1860",
  "event_name": "Election of 1860",
  "claims": [
    "The presidential election took place in November 1860.",
    "Lincoln's victory alarmed many Southern leaders.",
    "The election accelerated the movement toward secession."
  ],
  "temporal_details": {
    "date": "November 1860",
    "time": "",
    "place": ""
  },
  "tone": "Sympathetic"
}
```


This is an example event extraction for the Election of 1860, showing structured claims, date, and tone from a secondary source (Gutenberg).


```
{
  "doc_id": "loc_gettysburg_nicolay",
  "document_title": "Gettysburg Address (Nicolay Copy)",
  "source": "lincoln",
  "event": "gettysburg_address",
  "event_name": "Gettysburg Address",
  "claims": [
    "Lincoln spoke at the dedication of the Soldiers' National Cemetery at Gettysburg.
    "He emphasized the nation's founding principle of equality.",
    "He urged the living to continue the unfinished work of the fallen soldiers."
  ],
  "temporal_details": {
    "date": "November 19, 1863",
    "time": "",
    "place": "Gettysburg, Pennsylvania"
  },
  "tone": "Solemn"
}
```


This is another example of event extraction for the Gettysburg Address from a primary source (Lincoln's own speech).

PART 3 - LLM Judge Design and Validation

Part 3 has two segments:

- ➔ 3A - Design the event consistency judge
- ➔ 3B - Validate the judge (Prompt robustness, Self-consistency, Inter-rater)

3A - Designing the Judge

Here, part 3A asks to do the following:

- For each event
  - ➔ Compare claims from Lincoln vs other authors
  - ➔ Assign a consistency score (0-100)
  - ➔ Give examples of agreement
  - ➔ Give examples of contradiction, with types (factual/interpretive/omission)
  - ➔ Describe what's missing from Lincoln and what's missing from others
  - ➔ Compare the tone between Lincoln and others
- Output all of this in a structured JSONL file

Grouping claims by event and score

File: src/part3_eval/event_judge.py Function: group_claims_by_event

Here I am doing the following things:

- ➔ Grouping by event
- ➔ Separating "lincoln" vs "other"
- ➔ Deduplicating claims

```python
def group_claims_by_event(
    records: List[Dict[str, Any]]
) -> Dict[str, Dict[str, Any]]:
    grouped: Dict[str, Dict[str, Any]] = {}

    for rec in records:
        event_id = rec["event"]
        source = rec.get("source", "unknown")
        doc_id = rec["doc_id"]
        claims = rec.get("claims", [])

        if event_id not in grouped:
            grouped[event_id] = {
                "event_name": rec.get(
                    "event_name",
                    EVENTS.get(event_id).name if event_id in EVENTS else e
                ),
                "lincoln": {"doc_ids": [], "claims": []},
                "other": {"doc_ids": [], "claims": []},
                "unknown": {"doc_ids": [], "claims": []},
            }

        bucket = grouped[event_id].get(source, grouped[event_id]["unknown"
        if doc_id not in bucket["doc_ids"]:
            bucket["doc_ids"].append(doc_id)
        bucket["claims"].extend(claims)

    # Deduplicate claims per source
    for grp in grouped.values():
        for src in ["lincoln", "other", "unknown"]:
            seen = set()
            unique_claims = []
            for c in grp[src]["claims"]:
                c_norm = c.strip()
                if c_norm and c_norm not in seen:
```

The figure shows grouping extracted claims by event and source (Lincoln vs other authors), and deduplicating claims.

Judge prompt design

File: event_judge.py Function: build_judge_prompt

Here I am doing two things

I am giving the judge:

- ➔ event id + name + description
- ➔ Lincoln's claims
- ➔ Other's claims

I am instructing the judge to do:

- ➔ Output **overall_consistency** in 0-100
- ➔ Provide **agreement_examples**
- ➔ Provide **contradictions** with typed categories
- ➔ Fill missing-from-Lincoln/Others and tone

```
def build_judge_prompt(
    event_id: str,
    event_name: str,
    lincoln_claims: List[str],
    other_claims: List[str],
) -> Tuple[str, str]:
    event_cfg = EVENTS.get(event_id)
    description = event_cfg.description if event_cfg else ""

    system_prompt = (
        "You are a careful historical evaluator. Your task is to compare h
        "Abraham Lincoln's own writings describe an event versus how later
        "describe the same event. Be precise and fair. Use ONLY the claims
    )

    def fmt_list(lst: List[str]) -> str:
        if not lst:
            return "  (none)\n"
        return "\n".join(f"  - {c}" for c in lst)

    lincoln_block = fmt_list(lincoln_claims)
    other_block = fmt_list(other_claims)

    user_prompt = f"""
Event: {event_name} ({event_id})

Short description:
{description}

Set A: Claims from Abraham Lincoln's own writings
{lincoln_block}

Set B: Claims from other authors (histor ↓ s, biographers, etc.)
{other_block}
```

The figure shows Judge prompt specification for comparing Lincoln's claims vs late authors and producing structured consistency evaluations.

Output parsing and Normalization

File: event_judge.py **Function: safe_parse_judge_output**

What this function does:

- ➔ Robust JSON parsing
- ➔ Enforcing 0-100 range
- ➔ Normalizing **contradictions** into **{description, type}** with fallback inference

```python
def safe_parse_judge_output(output: str, event_id: str) -> Dict[str, Any]:
    text = output.strip()
    if text.startswith("```"):
        text = text.strip("`")
    start = text.find("{")
    end = text.rfind("}")
    if start != -1 and end != -1:
        text = text[start : end + 1]

    try:
        data = json.loads(text)
    except Exception:
        data = {}

    # Normalize
    if "event" not in data:
        data["event"] = event_id

    if not isinstance(data.get("overall_consistency"), (int, float)):
        data["overall_consistency"] = 50
    else:
        val = int(round(float(data["overall_consistency"])))
        data["overall_consistency"] = max(0, min(100, val))

    if not isinstance(data.get("agreement_examples"), list):
        data["agreement_examples"] = []

    # contradictions: list of {description, type}
    contras = data.get("contradictions")
    if not isinstance(contras, list):
        contras = []
    normalized_contras = []
    for c in contras:
        if not isinstance(c, dict):
            continue
```

The figure shows robust parsing and normalization of judge outputs, including clamping scores 0-100 and classifying contradiction types.

Main evaluation loop

Output: event_consistency.jsonl Function: evaluate_events

This function does the following things:

- ➔ Iterates over events
- ➔ Calls the LLM
- ➔ Write output to **data/evals/event_consistency.jsonl**

```python
def evaluate_events(grouped: Dict[str, Dict[str, Any]]) -> List[Dict[str,
    results: List[Dict[str, Any]] = []

    for event_id, grp in grouped.items():
        lincoln_claims = grp["lincoln"]["claims"]
        other_claims = grp["other"]["claims"]

        if not lincoln_claims and not other_claims:
            continue

        event_name = grp["event_name"]
        print(f"[info] Evaluating event {event_id} ({event_name})")

        system_prompt, user_prompt = build_judge_prompt(
            event_id, event_name, lincoln_claims, other_claims
        )

        raw_output = call_llm(system_prompt, user_prompt, temperature=0.2)
        parsed = safe_parse_judge_output(raw_output, event_id)

        parsed["event_name"] = event_name
        parsed["lincoln_doc_ids"] = grp["lincoln"]["doc_ids"]
        parsed["other_doc_ids"] = grp["other"]["doc_ids"]
        parsed["lincoln_claim_count"] = len(lincoln_claims)
        parsed["other_claim_count"] = len(other_claims)

        results.append(parsed)

    return results
```

Output

The output can be found under **data/evals/event_consistency.jsonl**


```
{
  "event": "gettysburg_address",
  "event_name": "Gettysburg Address",
  "overall_consistency": 92,
  "agreement_examples": [
    "Both Lincoln and later authors describe the speech as a dedication of the cemete
    "Both mention the theme of a new birth of freedom and the unfinished work of the
  ],
  "contradictions": [
    {
      "description": "Later authors sometimes portray the speech as instantly and uni
      "type": "omission"
    }
  ],
  "missing_from_lincoln": "Lincoln does not comment on how the speech was received by
  "missing_from_others": "Some later authors do not quote the full text and instead p
  "tone_comparison": "Lincoln's own tone is solemn and restrained, while later author
  "lincoln_doc_ids": ["loc_gettysburg_nicolay"],
  "other_doc_ids": ["gutenberg_12801", "gutenberg_14004"],
  "lincoln_claim_count": 5,
  "other_claim_count": 12
}
```


The figure shows the judge output for the Gettysburg Address, including consistency score, agreement, contradictions, and tone comparison

3B - Validation: Prompt Robustness, Self-Consistency, Inter-Rater

Once the judge was implemented in Part 3A, I treated it as a black-box evaluator and ran three families of experiments:

1. **Prompt robustness -** Does changing the prompt style change the answer?
2. **Self-consistency -** Does the same prompt give similar results across multiple runs?
3. **Inter-rater agreement (including kappa) -** Do different prompting strategies agree in a statistically meaningful way?

All of these experiments operate on the same underlying data: the event-level claims produced in Part 2 **data/events/event_extractions.jsonl**, grouped by event and source (Lincoln vs other authors)

3B.1 Prompt Robustness across strategies

My goal here was to test how sensitive the Judge is to prompt formulation:

- ➔ **Zero-shot:** Minimal instructions, no reasoning guidance.
- ➔ **Chain of Thought (cot):** "Think before giving the final JSON answer, considering each step in the process"
- ➔ **Few-shot:** It is an example comparison provided before the real data.

Implementation: In **event_judge_experiments.py**, the **build_strategy_prompt** function build system/user prompts for each strategy, and **run_prompt_robustness** iterates over all events and all three strategies

```
def run_prompt_robustness(grouped: Dict[str, Dict[str, Any]]) -> None:
    os.makedirs(os.path.dirname(PROMPT_ROBUST_OUT), exist_ok=True)
    strategies = ["zero_shot", "cot", "few_shot"]

    with open(PROMPT_ROBUST_OUT, "w", encoding="utf-8") as f_out:
        for event_id, grp in grouped.items():
            lincoln_claims = grp["lincoln"]["claims"]
            other_claims = grp["other"]["claims"]
            if not lincoln_claims and not other_claims:
                continue

            event_name = grp["event_name"]
            for strat in strategies:
                sys_prompt, user_prompt = build_strategy_prompt(
                    event_id, event_name, lincoln_claims, other_claims, strat
                )
                raw = call_llm(sys_prompt, user_prompt, temperature=0.2)
                score = extract_consistency_from_output(raw)

                record = {
                    "event": event_id,
                    "event_name": event_name,
                    "strategy": strat,
                    "overall_consistency": score,
                }
                f_out.write(json.dumps(record, ensure_ascii=False) + "\n")
```

The results are stored in **data/evals/prompt_robustness.jsonl,** one line per (event, strategy)

```
{"event": "gettysburg_address", "event_name": "Gettysburg Address", "strategy": "zero
{"event": "gettysburg_address", "event_name": "Gettysburg Address", "strategy": "cot",
{"event": "gettysburg_address", "event_name": "Gettysburg Address", "strategy": "few_
```


Observation: In various events, the three strategies produce similar consistency scores, typically within a few points of each other. This indicates that the prompt's appearance does not very influence the Judge's conclusion. Zero-shot, CoT, and few-shot variations all yield roughly the same understanding of Lincoln as later authors.

3B.2 Self-Consistency and Variance (Noise Analysis)

Goal: Measure how noisy the Judge is when rerun multiple times with the same prompt and a non-zero temperature. A stable judge should not fluctuate dramatically.

Implementation: In **run_self_consistency**, I do the following things:

- Fix the strategy to **cot.**
- For each event
  - ➔ Run the judge 5 times with **temperature = 0.7**
  - ➔ Extract the **overall_consistency** score each time
  - ➔ Compute:
    - **Mean**
    - **Min, max**
    - **Std (Standard deviation)**
    - **Coefficient_of_variation = std/mean**

```python
def run_self_consistency(grouped: Dict[str, Dict[str, Any]]) -> None:
    os.makedirs(os.path.dirname(SELF_CONSIST_OUT), exist_ok=True)

    with open(SELF_CONSIST_OUT, "w", encoding="utf-8") as f_out:
        for event_id, grp in grouped.items():
            lincoln_claims = grp["lincoln"]["claims"]
            other_claims = grp["other"]["claims"]
            if not lincoln_claims and not other_claims:
                continue

            event_name = grp["event_name"]
            sys_prompt, user_prompt = build_strategy_prompt(
                event_id, event_name, lincoln_claims, other_claims, strate
            )

            scores: List[int] = []
            for run_idx in range(5):
                raw = call_llm(sys_prompt, user_prompt, temperature=0.7)
                score = extract_consistency_from_output(raw)
                scores.append(score)

            mean_score = sum(scores) / len(scores)
            std = float(statistics.pstdev(scores)) if len(scores) > 1 else
            coef_var = float(std / mean_score) if mean_score > 0 else 0.0

            record = {
                "event": event_id,
                "event_name": event_name,
                "strategy": "cot",
                "runs": scores,
                "mean": mean_score,
                "min": min(scores),
                "max": max(scores),
                "std": std,
                "coefficient_of_variation": coef_var,
```

Output: data/evals/self_consistency.jsonl

```
{
  "event": "gettysburg_address",
  "event_name": "Gettysburg Address",
  "strategy": "cot",
  "runs": [90, 88, 89, 90, 89],
  "mean": 89.2,
  "min": 88,
  "max": 90,
  "std": 0.75,
  "coefficient_of_variation": 0.0084
}
```


Interpretation (variance).

- Standard deviation (std) measures how much the judge's scores vary around the average.
  ➔ A std of about 1 to 3 points on a 0 to 100 scale shows low noise.
- The coefficient of variation (CV = std/mean) adjusts this fluctuation; a very low CV means the relative instability is small.

Looking at different events, the judge's repeated scores show low variance and narrow ranges. This indicates that small numerical differences are mostly noise, but the overall qualitative judgment remains stable.

3B.3 Inter-Rater Agreement Across Strategies (Mean/Std/Range)

In addition to variance within a single strategy, I measured how much the three strategies (zero-shot, CoT, and few-shot) agree with each other.

Implementation: In `run_inter_rater_from_prompt_robustness`, I treat each strategy as a "rater" assigning a numerical score per event, and compute:

- `mean` – average consistency score across strategies.
- `std` – dispersion of scores across strategies.
- `range` – max − min score across strategies.

```python
def run_inter_rater_from_prompt_robustness() -> None:
    if not os.path.exists(PROMPT_ROBUST_OUT):
        return

    with open(PROMPT_ROBUST_OUT, "r", encoding="utf-8") as f:
        rows = [json.loads(line) for line in f if line.strip()]

    by_event: Dict[str, Dict[str, Any]] = {}
    for r in rows:
        ev = r["event"]
        by_event.setdefault(
            ev,
            {
                "event": ev,
                "event_name": r.get("event_name", ev),
                "scores": {},
            },
        )
        by_event[ev]["scores"][r["strategy"]] = r["overall_consistency"]

    with open(INTER_RATER_OUT, "w", encoding="utf-8") as f_out:
        for ev, rec in by_event.items():
            scores_dict = rec["scores"]
            scores_list = list(scores_dict.values())
            if not scores_list:
                continue

            mean_score = sum(scores_list) / len(scores_list)
            std = float(statistics.pstdev(scores_list)) if len(scores_list
            score_range = max(scores_list) - min(scores_list)

            out = {
                "event": ev,
                "event_name": rec["event e"],
                "strategy_scores": scores_dict,
```

Output: data/evals/inter_rater.jsonl


```
{
  "event": "gettysburg_address",
  "event_name": "Gettysburg Address",
  "strategy_scores": {
    "zero_shot": 90,
    "cot": 88,
    "few_shot": 87
  },
  "mean": 88.33,
  "std": 1.25,
  "range": 3
}
```

Interpretation.

- Small std and range indicate that different prompting strategies come to very similar conclusions about consistency.
- This is a numeric way of saying: *"Changing the prompt style does not fundamentally change the story the Judge tells."*

3B.4 Cohen's Kappa: Categorical Inter-Rater Reliability

To complement raw dispersion metrics, I computed **Cohen's κ**, a standard inter-rater agreement metric from statistics.

Because κ is defined over **categorical** labels, I first converted the numeric consistency scores into three categories:

- **high**: score ≥ 80
- **medium**: $50 \leq$ score < 80
- **low**: score < 50

This mapping is implemented in:

```python
def categorize_score(score: int) -> str:
    if score >= 80:
        return "high"
    elif score >= 50:
        return "medium"
    else:
        return "low"
```

For each event with all three strategies present, I have:

- Converted zero-shot, CoT, and few-shot scores into labels.
- Built three label vectors across events:
    - `zero_shot`: ["high", "medium", …]
    - `cot`: ["high", "medium", …]
    - `few_shot`: ["high", "medium", …]
- Computed Cohen's κ pairwise between strategies using a small custom implementation:

```python
def cohen_kappa(labels_a: List[str], labels_b: List[str]) -> float:
    """
    κ = (P_o − P_e) / (1 − P_e)
    where:
      − P_o is observed agreement
      − P_e is expected agreement by chance
    """
    assert len(labels_a) == len(labels_b)
    n = len(labels_a)
    if n == 0:
        return 0.0

    categories = sorted(set(labels_a) | set(labels_b))
    if not categories:
        return 0.0

    agree = sum(1 for a, b in zip(labels_a, labels_b) if a == b)
    p_o = agree / n

    freq_a = {c: 0 for c in categories}
    freq_b = {c: 0 for c in categories}
    for a in labels_a:
        freq_a[a] += 1
    for b in labels_b:
        freq_b[b] += 1

    p_e = 0.0
    for c in categories:
        p_e += (freq_a[c] / n) * (freq_b[c] / n)

    if p_e == 1.0:
        return 1.0
    return (p_o − p_e) / (1.0 − p_e) if (1.0 − p_e) > 0 else 0.0
```

Output: data/evals/kappa_inter_rater.jsonl

```
{
  "events": ["election_1860", "fort_sumter", "gettysburg_address", ...],
  "category_labels": {
    "zero_shot": ["high", "high", "high", ...],
    "cot":       ["high", "medium", "high", ...],
    "few_shot":  ["high", "medium", "high", ...]
  },
  "kappa": {
    "zero_vs_cot": 0.8,
    "zero_vs_few": 0.75,
    "cot_vs_few":  0.9
  }
}
```

Interpretation (κ).

- Cohen's κ compares observed agreement (P_o) with chance agreement (P_e).
- $\kappa \approx 0 \rightarrow$ agreement no better than chance.
- $\kappa > 0.6 \rightarrow$ substantial agreement (Landis & Koch).
- $\kappa > 0.8 \rightarrow$ near-perfect agreement.

In my runs, κ values between prompting strategies were in the substantial to near-perfect range, which means: even when we compress scores into coarse categories, different prompt variants still agree on which events are "high", "medium", or "low" consistency. This strengthens the case that the Judge is not just noisy, but genuinely stable.

3B.5 Distinguishing Noise vs Insight (Hallucination vs Real Signal)

Finally, I manually inspected several Judge outputs to check whether deviations were:

Noise / hallucination - the Judge fabricates details not supported by claims.

Real insight- The Judge duly brings to light differences between Lincoln and later authors.

Two patterns emerged:

1. **Noise example (hallucination)**

   In the case of one event, a CoT run reported that the Gettysburg Address "was immediately hailed by newspapers as a masterpiece," even though neither Lincoln's own text nor the secondary claims explicitly mentioned press reception. This is a classic LLM hallucination: historically plausible but unsupported by the input claims.

2. **Insight Example (actual variation)**

   For the same event, multiple judge runs in consensus recorded the following:

- Lincoln's text is about sacrifice, equality, and "unfinished work."
- Later authors stress the speech's mythic status and its long-term symbolic significance.

This is a real interpretive insight: the Judge is correctly noticing that the historians frame the address as iconic in ways that Lincoln himself, writing in the moment, obviously does not.

By integrating: Variance metrics (std, CV) Agreement metrics (mean, range, κ), and Qualitative check of particular contradictions and "missing" fields, I can differentiate random LLM noise from actual historical observations. No single run is treated as "truth," while the behavior of the ensemble plus statistical profiles gives confidence that the Judge captures real patterns rather than just hallucinating.