Integrating Re-prompting Pipeline into the Python SDK

July 24, 2025

Link to the PR: Coming soon!

Author: Aanya Shah

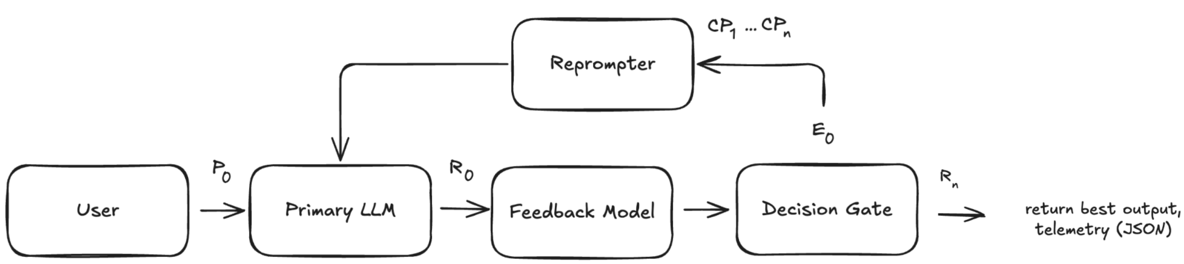
Objective

Integrate automated re-prompting pipeline into the Python SDK as a configurable API to provide users with a lightweight, model-agnostic tool to automatically improve instruction adherence and reduce hallucinations.

Motivation

Extracting a fully instruction adherent output from LLM, especially small LLMs with fewer parameters, can be tricky and require multiple iterations and hallucinations can be easily interpreted as fact. By adding a re-prompting loop to identify and correct instruction and groundedness violations, we can increase the instruction following rate by an average of ~22% and enable smaller models to surpass performance of GPT-4o. A more in-depth description of the re-prompting pipeline and testing can be found in this blogpost: [“Re-Prompting: A Smarter Loop for Smarter Models.”]

Re-prompting Pipeline

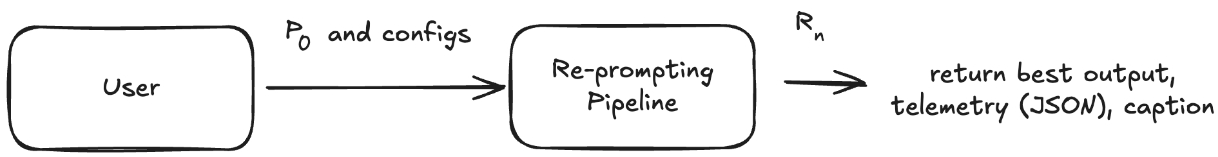


The interaction unfolds as a deterministic loop whose goal is to converge on an error-free answer with the fewest primary LLM passes:

1. **Initial Draft (P_0 → R_0).** The user’s prompt is forwarded untouched to the Primary LLM, which produces Draft R_0.
2. **Rapid Evaluation (R_0 → E_0).** The Feedback Model (IFE) inspects R_0 on groundedness, toxicity, and instruction adherence and emits Error Report E_0. Each entry lists the violated instruction, follow probability, and an explanation.
3. **Decision Gate.** If there are no instruction violations, R_n is returned. If a hard cap of N iterations is reached if the latency budget is exceeded (or less than 25% remains), the draft among {R_0 … R□} with the least failed instructions or lowest residual error score is surfaced to the user along with optional metadata (e.g., “2 iterations, 750 ms”). Otherwise the Coordinator proceeds to the next step.

4. **Corrective Prompt Generation.** The Re‑Prompter synthesizes Corrective Prompt P_1 that embeds the user's original message plus a distilled list of violated instructions and explanations.
5. **Iterative Refinement.** The Corrective Prompt is sent back to the Primary LLM and the loop continues.

User Experience



Function Signature:

[redacted]

Return Values:

A dictionary with:

- **best_response (str)** — Final LLM output selected by the pipeline.
- **telemetry (dict, optional)** — Per‑iteration JSON metadata (if return_telemetry=True)
- **summary (str, optional)** — Human‑readable summary (if return_aimon_summary=True).
  - E.g.: 2 iterations, 0 failed instructions
  - E.g.: 3 iterations, 1 failed instructions

Parameters

Parameters Overview:

(* = required)

Name	Type	Description
llm_fn*	Callable [[str, str, str],[str]]	**Required**. Function to call any primary LLM that takes in context, user_query, system prompt and returns a string output.
user_query*	str	**Required**. user question

system_prompt	str	
context	str	Context for the query. Only accepts raw text (str)
user_instructions	List[str]	Guidelines for the model (given to the model in the prompt and used by IFE to evaluate adherence)
reprompting_config	RepromptingConfig	Optional. Advanced settings for the pipeline.

RepromptingConfig Overview:

Name	Type	default	Description
aimon_api_key	str	env:AIMON_API_KEY	API key to call IFE feedback model
publish	bool	no	Flag indicating whether to publish the results to app.aimon.ai
max_iterations	int	2	Max number of LLM calls (1 initial + reprompts).
return_telemetry	bool	no	Return a json blob with per-iteration metadata to trace re-prompting
return_aimon_summary	bool	no	Returns a short caption about re-prompting metadata (e.g., “2 iterations, 0 failed instructions”)
latency_limit_ms	int	none	Abort loop and return current best response if total latency exceeds this.
model_name	str	Defaults to a string based on "aimon-react-model" concatenated with a random string.	Model name for telemetry.
application_name	str	Defaults to a string based on	Name of the Application name for

		"aimon-react-application" concatenated with a random string.	telemetry.
user_model_max_retries	int	1	Exponential backoff based retries the given number of times if llm_fn fails
feedback_model_max_retries	int	1	Exponential backoff based retries the given number of times if AIMon Detect fails

Parameters and Config Breakdown

Llm_fn: User-provided function that takes a single str prompt and returns a str output.
Wrapped in try/except. On failure, the pipeline explicitly throws an error so the user can decide next steps. We will also implement an exponential backoff based retry up to user_model_max_retries. Empty or invalid outputs after retries are treated as failures.

Llm_fn: user-provided llm_fn is defined as a Callable that outputs a string and accepts:

- **recommended_prompt_template:** string.Template
- **system_prompt:** str
- **context:** str
- **user_query:** str

The initial and corrective prompt will both be provided by the reprompting pipeline in Template form so the user will have to substitute system_prompt, context, and user_query placeholders in the Template with the values of the function parameters. These fields can be made optional, but right now run_reprompting_pipeline provides blank placeholders if the user doesn't specify a system_prompt or context which can be passed into the llm_fn.
This design requires users to:

- Handle their own LLM calls.
- Build the initial prompt by either concatenating system_prompt, context, and user_query as they please or substituting appropriate values into the provided recommended_prompt_template.

Example implementation:

```
TOGETHER_API_KEY = os.environ.get("TOGETHER_API_KEY")
client = Together(api_key=TOGETHER_API_KEY)

def my_llm(recommended_prompt_template: Template, system_prompt, context, user_query) -> str:
```

```
    # substitute placeholders in the pipeline-provided template with
appropriate values
    filled_prompt = recommended_prompt_template.substitute(
        system_prompt=system_prompt,
        context=context,
        user_query=user_query
    )

    # replace this block with any LLM call you want. (OpenAI, Claude,
HuggingFace, etc.)
    response = client.chat.completions.create(
        model="google/gemma-3n-E4B-it", # this can be any Together-hosted
model (e.g., 'mistralai/Mistral-7B-Instruct-v0.2')
        messages=[{"role": "user", "content": filled_prompt}],
        max_tokens=256, # increase for longer outputs
        temperature=0 # raise for more creative outputs
    )

    # extract and return a string output
    output = response.choices[0].message.content
    return output
```

Context:
Takes in str. Users can extract context using RAG systems like LlamaIndex or Langchain independent of the re-prompting system, normalize it to a str, and pass in their retrieved context from any source.

Latency_limit_ms:
At the start of each iteration, we check the remaining latency budget. If at least 25% of the budget is left, the loop continues to the next iteration. Otherwise, it terminates and returns the last valid response with a caption (e.g., "[Latency limit exceeded on iteration N]"). Telemetry logs stop_reason = "latency_limit_exceeded".

Monitoring and Telemetry

Return_telemetry:
Each iteration outputs a JSON blob with the following information:

- Model_name
- application_name
- iteration

- cumulative_latency_ms
- groundedness_score
- instruction_adherence_score
- residual_error
- failed_instructions_count
- Stop_reason (one of the following)
  - All_instructions_adhered
  - Max_iterations_reached
  - Latency_limit_exceeded
  - Llm_call_failed (explicit error thrown)
  - Feedback_model_failed (explicit error thrown)
  - unknown_error
- prompt
- response_text
- response_feedback: IFE model feedback on failed instructions

Should I implement native python logging (e.g.: import logging, logger = logging.getLogger()) and log errors, warnings, or actions or is the json blob return value and in-memory telemetry sufficient? Right now, some errors are dealt with silently and not surfaced to the user.

LlamaIndex and Langchain integration

User-Managed Retrieval

- The user is responsible for retrieving relevant context before calling the pipeline.
- They convert the retrieved content into a str and pass it into the context parameter of run_reprompting_pipeline().

This is super simple for the pipeline as no added retrieval logic will need to be implemented and the pipeline remains framework agnostic. Users retain full control over retrieval configs / source but it requires more effort on their end.

Alternatives Considered

2. **Option 2: Retriever in Config and Per-Call Toggle**

The user sets indices and initializes a llamaindex_retriever or langchain_retriever and a top_k value that is passed in the RepromptingConfig. For each call of the re-prompting pipeline, they pass in a bool use_llamaindex or use_langchain. If either are true, the re-prompting pipeline retrieves the context and uses the result as context. If both are true, the pipeline concatenates context from both retrievals. If use_llamaindex or use_lanchain is true and no or an invalid retriever is passed in, the pipeline can terminate with an error message or run without context.

Which do you recommend? There is an optional override to pass llamaindex_retriever/langchain_retriever and top_k to run_reprompting_pipeline and that one will be used instead of the one in RepromptingConfig.

- Users set up persistent retrievers (llamaindex_retriever or langchain_retriever) and a default top_k value in RepromptingConfig.
- For each call to run_reprompting_pipeline(), they can pass use_llamaindex=True and/or use_langchain=True.
- When either is set, the pipeline automatically performs retrieval.
- **Optional per-call overrides:** Users can override llamaindex_retriever, langchain_retriever, and top_k in the run_reprompting_pipeline() call.

If retrieval fails, should the pipeline

- Terminates with a clear error, **or**
- Proceeds with no context?

3. **Option 3: Per-call Retrieval Config**
Users pass llamaindex_retriever/langchain_retriever, use_llamaindex/use_langchain, and top_k for each pipeline call which gives them more flexibility versus Option 2. How it works in the pipeline / failure modes are the same as Option 2.

Example Implementations

[OLD] Llm_fn (Misral7B via TogetherAI):

```
from together import Together

# Initialize Together client
client = Together(api_key="YOUR_TOGETHER_API_KEY")

# Define llm_fn
def my_llm(prompt: str) -> str:
   """Calls a Mistral model and returns the generated text."""
   response = client.chat.completions.create(
       model="mistralai/Mistral-7B-Instruct-v0.2",  # Any Together-hosted model
       messages=[{"role": "user", "content": prompt}],
       max_tokens=512,
       temperature=0
   )
   return response.choices[0].message.content
```

LlamaIndex Integration Example:

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
from aimon.reprompting_api.runner import run_reprompting_pipeline
from aimon.reprompting_api.config import RepromptingConfig
```

```python
import os
from openai import OpenAI

# --- SETUP LLM + CONFIG ---
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
AIMON_API_KEY = os.getenv("AIMON_API_KEY")
client = OpenAI(api_key=OPENAI_API_KEY)

def my_llm(prompt: str) -> str:
   """LLM wrapper for use in the pipeline."""
   response = client.chat.completions.create(
       model="gpt-4o-mini",
       messages=[{"role": "user", "content": prompt}],
       max_tokens=500,
       temperature=0
   )
   return response.choices[0].message.content

config = RepromptingConfig(
   aimon_api_key=AIMON_API_KEY,
   max_iterations=2,
   return_telemetry=True,
   return_aimon_summary=True
)

# --- SETUP LLAMAINDEX FOR RETRIEVAL ---
# Load documents (replace with your source: PDFs, DBs, etc.)
documents = SimpleDirectoryReader(input_dir="./docs").load_data()

# Build index
index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine()

# --- USER MANAGED RETRIEVAL ---
user_query = "Summarize the company's data retention policies."
retrieved_nodes = query_engine.query(user_query)

# Convert retrieved nodes to a plain string for the pipeline
context = "\n\n".join([str(node) for node in retrieved_nodes])

# --- RUN THE PIPELINE ---
instructions = [
```

```
    "Answer in 3 concise bullet points.",
    "Ensure your response is based only on the provided context.",
    "Avoid speculative or vague language."
]

response = run_reprompting_pipeline(
    user_query=user_query,
    context=context,  # User-managed retrieval result
    llm_fn=my_llm,
    user_instructions=instructions,
    reprompting_config=config
)

# --- DISPLAY RESULTS ---
print("\n=== BEST RESPONSE ===")
print(response["best_response"])

if "summary" in response:
    print("\n=== SUMMARY ===")
    print(response["summary"])

if "telemetry" in response:
    print("\n=== TELEMETRY ===")
    for entry in response["telemetry"]:
        print(entry)
```

LangChain Integration Example:

```
from langchain_openai import OpenAIEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from aimon.reprompting_api.runner import run_reprompting_pipeline
from aimon.reprompting_api.config import RepromptingConfig
import os
from openai import OpenAI

# --- SETUP LLM + CONFIG ---
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
AIMON_API_KEY = os.getenv("AIMON_API_KEY")
client = OpenAI(api_key=OPENAI_API_KEY)
```

```python
def my_llm(prompt: str) -> str:
   """LLM wrapper for use in the pipeline."""
   response = client.chat.completions.create(
       model="gpt-4o-mini",
       messages=[{"role": "user", "content": prompt}],
       max_tokens=500,
       temperature=0
   )
   return response.choices[0].message.content

config = RepromptingConfig(
   aimon_api_key=AIMON_API_KEY,
   max_iterations=2,
   return_telemetry=True,
   return_aimon_summary=True
)

# --- SETUP LANGCHAIN FOR RETRIEVAL ---
# Load and split documents
loader = TextLoader("./docs/policies.txt")
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
split_docs = splitter.split_documents(docs)

# Create embeddings and index
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
vectorstore = FAISS.from_documents(split_docs, embeddings)

# User query
user_query = "Summarize the company's data retention policies."

# Retrieve top-k documents
retrieved_docs = vectorstore.similarity_search(user_query, k=3)

# Convert retrieved docs to a plain string
context = "\n\n".join([doc.page_content for doc in retrieved_docs])

# --- RUN THE PIPELINE ---
instructions = [
   "Answer in 3 concise bullet points.",
   "Ensure your response is based only on the provided context.",
```

```
    "Avoid speculative or vague language."
]

response = run_reprompting_pipeline(
    user_query=user_query,
    context=context,  # User-managed retrieval result
    llm_fn=my_llm,
    user_instructions=instructions,
    reprompting_config=config
)

# --- DISPLAY RESULTS ---
print("\n=== BEST RESPONSE ===")
print(response["best_response"])

if "summary" in response:
    print("\n=== SUMMARY ===")
    print(response["summary"])

if "telemetry" in response:
    print("\n=== TELEMETRY ===")
    for entry in response["telemetry"]:
        print(entry)
```

Rough Example Implementation:

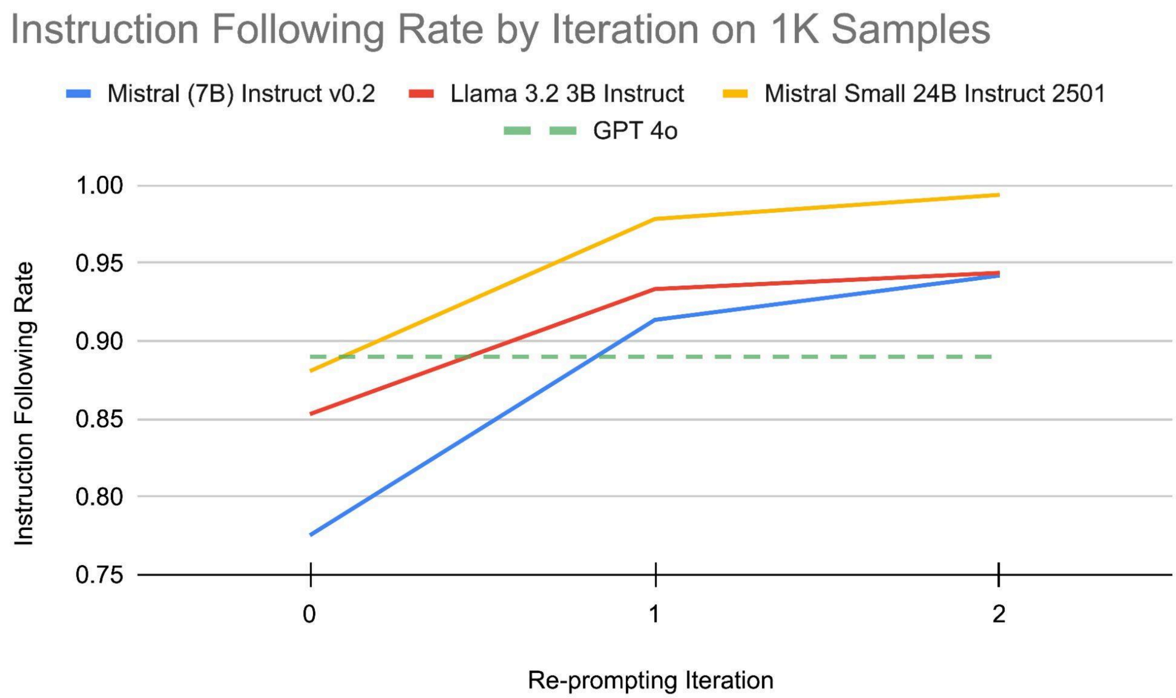
```
from together import Together
from aimon.reprompting_api.runner import run_reprompting_pipeline
from aimon.reprompting_api.config import RepromptingConfig

# Initialize Together client
client =
Together(api_key="8b6726e35a842117f91077ca78fc69e1ee285c998592fd8356bd4123a63378a1")

# Define llm_fn
def my_llm(prompt: str) -> str:
    response = client.chat.completions.create(
        model="mistralai/Mistral-7B-Instruct-v0.2",  # Any Together-hosted model
        messages=[{"role": "user", "content": prompt}],
        max_tokens=512,
```

```
        temperature=0
    )
    return response.choices[0].message.content

# Initialize configuration and components
config = RepromptingConfig(
    aimon_api_key="998e211045c1d9e5b1fc0fa9e9e001be684596d8d529952c088eb1627480529c",
    publish=True,
    return_telemetry=True,
    return_aimon_summary=True,
    application_name="api_test",
    tokenizer_fn = lambda text: 1 / 0  # Will cause ZeroDivisionError
)

context = """SecureCloud offers encrypted file storage with two-factor authentication
and regular security audits to protect user data."""
user_instructions = [
    "Be concise",
    "Use informal language."
]

result = run_reprompting_pipeline(
        user_query=user_query,
        context=context,
        llm_fn=my_llm,
        reprompting_config=config,
        user_instructions=user_instructions
)

print("\nRe-prompted response:")
print(result["best_response"])
print(result.get("telemetry"))
print(result.get("summary"))
```

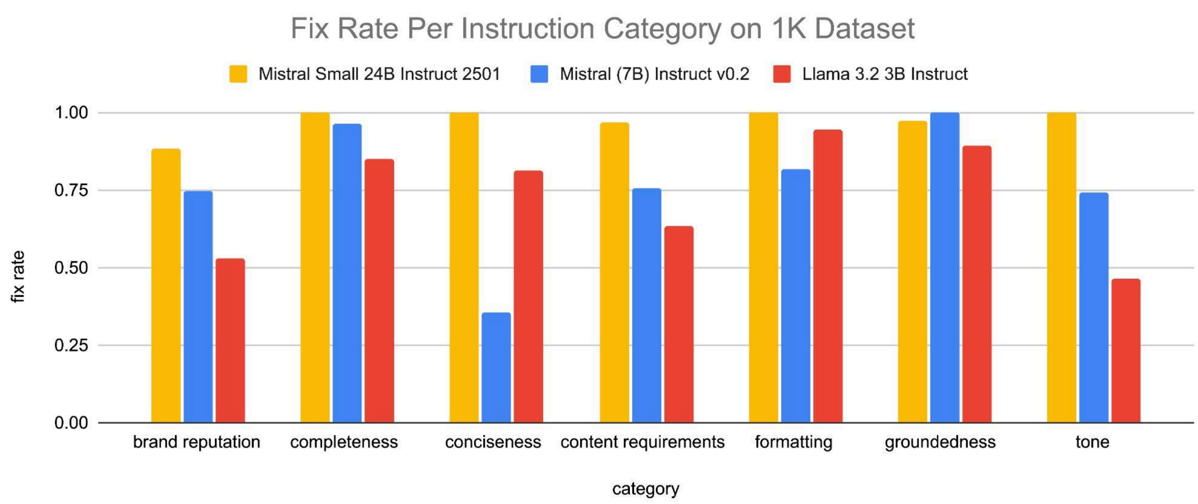
Re-prompting Experiment Results

Overall Results

While all models initially underperformed GPT-4o, re-prompting allowed every model to surpass GPT-4o's baseline.



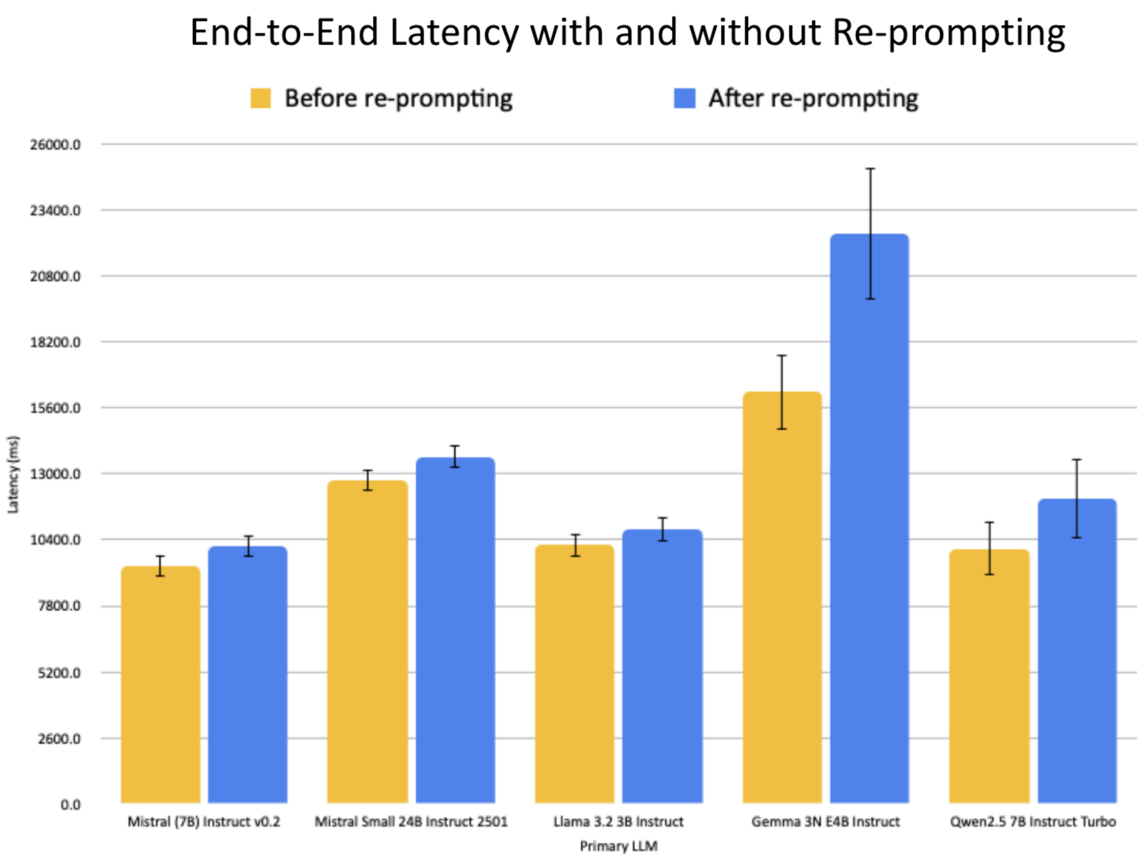
Instruction-Level Analysis



Re-prompting was especially effective at resolving groundedness violations (hallucinations), thanks to IA2's precise token-level feedback. However, more subjective instructions like conciseness proved harder to fix consistently.

Latency

This graph shows average end-to-end latency (with 95% confidence intervals) across all samples, comparing outputs without and with use of the re-prompting pipeline. The "after re-prompting" values reflect the overall average across all queries, including those resolved on the first pass (and thus requiring no re-prompting) and those requiring multiple iterations.



On average, samples that required re-prompting exhibited a 5,121.2 ms increase in latency (roughly a 47% increase), reflecting the extra workload of recalling the primary LLM up to two additional times. However, because many queries will adhere to instructions on the first pass (requiring no re-prompting), the overall impact on large-scale workloads is much less pronounced. Additionally, the 95% confidence intervals for almost all models overlap, indicating that this added latency is statistically negligible when viewed across many LLM calls.

Limitations

- Relies on clear, deterministic, and realistic instructions. Vague or contradictory constraints are difficult to fix.
- Subjective attributes (tone, conciseness) show inconsistent improvement.
- Relies on IA model's ability to accurately identify failures
- Adds latency and increases cost; trade-offs must be considered for real-time or budget-sensitive applications.

Milestones

- [ ] Determine configs / aspects of the pipeline to allow users to alter
- [ ] Refactor into run_reprompting_pipeline
- [ ] Clean up RepromptingConfig
- [ ] Add latency limit logic
- [ ] Gracefully handle failures
- [ ] Write tests in Collab Notebook
- [ ] PR and review
- [ ] API description on Docs with clear implementation guidelines, use cases, etc.