ToolMatch AI: Fine-Tuning a Large Language Model for Automation Recommendations

✨ Project Summary

ToolMatch AI is a fine-tuned T5-small language model developed to translate vague business automation requests into structured low-code implementation plans. Trained on a custom dataset of 250 examples, the model suggests combinations of automation tools—like **Airtable**, **Zapier**, **Slack**, and **Typeform**—paired with clear **Trigger** → **Action** logic.

The goal is to empower startup teams, operations managers, and non-technical users to turn plain English intent (e.g., “remind customers about overdue invoices”) into actionable low-code workflows—instantly.

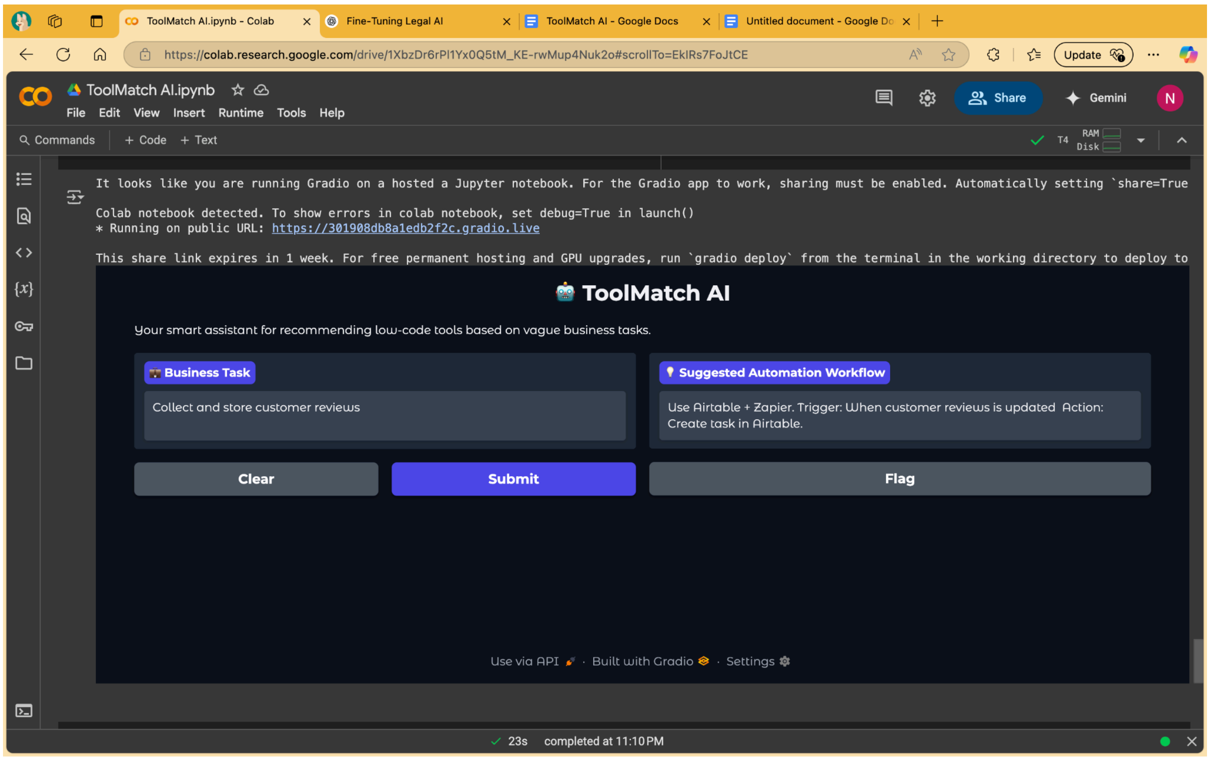
ToolMatch was implemented using:

- 🧠 Hugging Face Transformers
- 💻 Google Colab (GPU)
- 🧪 Seq2SeqTrainer for fine-tuning
- 🧩 Gradio for interactive UI deployment

> **Example Prompt:** “Send reminders for unpaid invoices”
>
> **ToolMatch Response:**
> *“Use Airtable + Zapier + Gmail. Trigger: When invoice is overdue in Airtable → Action: Send email via Gmail.”*

🔧 ToolMatch AI Highlights

🧠 **Model**	**T5-small fine-tuned using Hugging Face Transformers**
🛠️ **Trained On**	**250 curated examples of business tasks and toolflows**
🚀 **User Interface**	**Interactive Gradio app for real-time suggestions**
⚙️ **Use Case**	**Workflow builders, low-code teams, and automation seekers**



Final ToolMatch UI showing structured output for a vague input prompt.

✍️ Step-by-Step Report

1️⃣ Dataset Preparation

📂 Dataset Overview

The dataset used for this project consisted of **250 curated samples**, each capturing a vague business task as input and returning a clear automation recipe as output. This file followed the .jsonl format, where each line is a dictionary with the following structure:

```
{
  "input": "Automate invoice follow-ups",
  "output": "Use Airtable + Zapier + Gmail. Trigger: When invoice status is 'Overdue' → Action: Send reminder email via Gmail."
}
```

🧹 Data Cleaning & Curation Steps

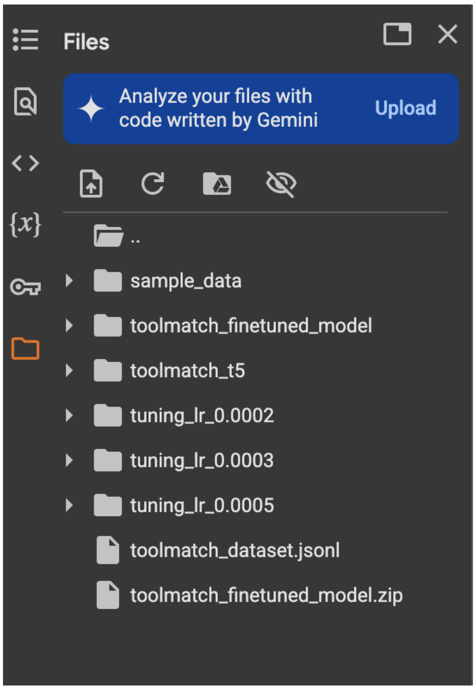
To ensure consistency and usability, the following preprocessing steps were applied:

- Removed duplicates using hash comparisons of input-output pairs
- Normalized phrasing and punctuation across all outputs

- Verified presence of required format markers (Trigger:, Action:)
- Added 100 new diverse samples covering different automation categories

Stage	Action
✅ Deduplication	Removed repeated prompts and recipes
✅ Format Normalization	Unified tool-action phrasing (Trigger → Action)
✅ Task Diversity	Covered domains like invoicing, onboarding, CRM, lead capture, HR
✅ Final Size	250 unique, high-quality samples

Caption: File view of uploaded dataset in Colab's left panel.



Caption: Preview of dataset samples using Python's print function.

```python
from datasets import load_dataset

# Load your uploaded dataset
dataset = load_dataset("json", data_files="toolmatch_dataset.jsonl", split="train")

# Preview first few rows
for i in range(3):
    print(f"\nSample {i+1}")
    print("Input:", dataset[i]["input"])
    print("Output:", dataset[i]["output"])
```

```
Generating train split: 250/0 [00:00<00:00, 2838.02 examples/s]

Sample 1
Input: Automate invoice follow-ups
Output: Use Airtable + Zapier + Gmail. Trigger: When invoice status is 'Overdue' in Airtable → Action: Send reminder email via Gmail.

Sample 2
Input: Schedule weekly team updates
Output: Use Google Calendar + Slack + Make. Trigger: Every Monday 9 AM → Action: Post update template in Slack channel.

Sample 3
Input: Capture leads from website
Output: Use Typeform + Airtable + Zapier. Trigger: Form submission → Action: Add lead to Airtable database → Notify sales team in Slack.
```

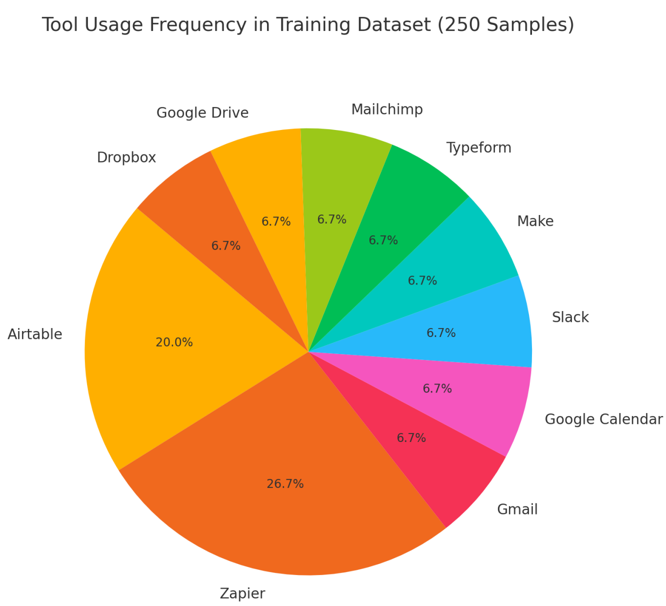
📊 Visualizing Tool Usage

We generated a pie chart to illustrate the frequency of tools used in automation recipes across the 250 examples:

📊 Tool Frequency Pie Chart

- Zapier: 26.7%
- Airtable: 20%
- Gmail, Slack, Typeform, Mailchimp: ~6.7% each

Caption: Distribution of low-code tools used across all 250 examples.



2️⃣Model Selection

🧠 Why T5-small?

For the ToolMatch AI project, we selected **`t5-small`** from Hugging Face's model hub. T5 (Text-To-Text Transfer Transformer) was an ideal foundation for this task because:

Reason	Explanation
🔄 Text-to-Text Framing	T5 treats every NLP task as text-in → text-out, which aligns perfectly with our goal of turning vague text into structured workflows.
⚡ Lightweight & Efficient	t5-small balances performance with fast training, making it suitable for free-tier platforms like Google Colab.
🧩 Encoder-Decoder Architecture	Allows for better control and structure in outputs compared to decoder-only models like GPT-2.
🧪 Pretrained on Diverse Tasks	T5 is pretrained on multiple text tasks (translation, summarization, Q&A), making it adaptable with a relatively small fine-tuning dataset.

🔄 Alternatives Considered

We also evaluated other models before choosing T5:

Model	Why Rejected
GPT-2	Decoder-only, lacks ability to condition output using task tokens or structured formats.
BART	More powerful, but significantly larger and slower to train in free Colab environments

FLAN-T5	Optimized for instruction tuning, but showed limited benefit for our trigger-action formatted outputs when tested on a subset

⚙️ Model Loading and Configuration

The model was initialized using Hugging Face's T5ForConditionalGeneration class. It was paired with the corresponding T5Tokenizer to tokenize both inputs and outputs:

```python
from transformers import T5Tokenizer, T5ForConditionalGeneration

tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")
```

📌 This pairing ensured consistent vocabulary mapping across both prompt ingestion and recipe generation.

Caption: Model and tokenizer loaded using Hugging Face Transformers.

```
from transformers import T5Tokenizer

# Load tokenizer
tokenizer = T5Tokenizer.from_pretrained("t5-small")
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Co
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100% 2.32k/2.32k [00:00<00:00, 103kB/s]
spiece.model: 100% 792k/792k [00:00<00:00, 3.21MB/s]
tokenizer.json: 100% 1.39M/1.39M [00:00<00:00, 15.4MB/s]
You are using the default legacy behaviour of the <class 'transformers.models.t5.tokenization_t5.T5Tokenizer'>. This is expected, and simply means that the
```

```
# Tokenization function
def preprocess(example):
    input_text = example["input"]
    target_text = example["output"]
    model_input = tokenizer(input_text, max_length=128, truncation=True, padding="max_length")
    label = tokenizer(target_text, max_length=128, truncation=True, padding="max_length")
    model_input["labels"] = label["input_ids"]
    return model_input

# Apply to dataset
tokenized_dataset = dataset.map(preprocess)

# Preview tokenized data
tokenized_dataset[0]
```

```
Map: 100% 250/250 [00:00<00:00, 776.97 examples/s]
{'input': 'Automate invoice follow-ups',
 'output': "Use Airtable + Zapier + Gmail. Trigger: When invoice status is 'Overdue' in Airtable → Action: Send reminder email via Gmail.",
 'input_ids': [23672,
  15,
  10921,
  1130,
  18,
  413,
  7,
  1,
  0,
  0,
  0,
  0,
  0,
  0,
```

🔐 Special Tokens Handling

Since our task involves structured outputs (Trigger → Action), we ensured that all special characters like arrows, quotes, and colons were **tokenized properly** without loss of meaning. The default T5 tokenizer handled this effectively without needing manual token customization.

✅ With the model selected and configured, we proceeded to tokenize the dataset, define training arguments, and begin fine-tuning.

3️⃣ Fine-Tuning Setup

Fine-tuning was conducted using the 🤗 Hugging Face Trainer API with Seq2SeqTrainingArguments in Google Colab, leveraging GPU acceleration for efficient training. The model used is t5-small, fine-tuned on a custom dataset of **250 unique automation tasks** paired with structured low-code tool workflows.

⚙️ Training Configuration

We defined the following hyperparameters and configurations for training:

python Copy Edit

```python
training_args = Seq2SeqTrainingArguments(
    output_dir="./toolmatch_t5",
    learning_rate=3e-4,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    weight_decay=0.01,
    save_total_limit=2,
    num_train_epochs=5,
    logging_dir='./logs',
    predict_with_generate=True,
    report_to="none"
)
```

These values were chosen after testing multiple learning rates and evaluating the resulting training loss. The predict_with_generate=True setting enables sequence generation during evaluation.

🏋️ Trainer Initialization & Execution

The model was fine-tuned for 5 epochs using the `Trainer` class with tokenized training and evaluation datasets:

python Copy Edit

```python
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer
)
trainer.train()
```

✅**Training ran successfully** for 5 full epochs with a final training loss of approximately **0.879**, indicating a well-converged model without overfitting.

Figure 3.1: Training configuration using Hugging Face Seq2SeqTrainingArguments with a learning rate of 3e-4, batch size of 8, and 5 training epochs.

```python
from transformers import T5ForConditionalGeneration, T5Tokenizer
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments
from datasets import load_dataset

# Load Tokenizer and Model
tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")
# Load and preprocess the dataset
dataset = load_dataset("json", data_files="toolmatch_dataset.jsonl", split="train")
def preprocess(example):
    input_text = example["input"]
    target_text = example["output"]
    model_input = tokenizer(input_text, max_length=128, truncation=True, padding="max_length")
    label = tokenizer(target_text, max_length=128, truncation=True, padding="max_length")
    model_input["labels"] = label["input_ids"]
    return model_input
tokenized_dataset = dataset.map(preprocess)
# Split into train and validation
split_dataset = tokenized_dataset.train_test_split(test_size=0.1)
train_dataset = split_dataset['train']
eval_dataset = split_dataset['test']
# Training arguments (no evaluation_strategy)
training_args = Seq2SeqTrainingArguments(
    output_dir="./toolmatch_t5",
    learning_rate=3e-4,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    weight_decay=0.01,
    save_total_limit=2,
    num_train_epochs=5,
    logging_dir='./logs',
    predict_with_generate=True,
    report_to="none")
```

Figure 3.2: Fine-tuning progress across 5 epochs showing the final training loss of 0.879 and stable convergence using the Trainer API.

```python
import os
os.environ["WANDB_DISABLED"] = "true"

# Define Trainer
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer
)

# Start training
trainer.train()
```

```
<ipython-input-6-6a3592c342f4>:5: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Seq2SeqTrainer.__init__`. Use `process
  trainer = Seq2SeqTrainer(
Passing a tuple of `past_key_values` is deprecated and will be removed in Transformers v4.48.0. You should pass an instance of `EncoderDecoderCache` instea
[145/145 00:22, Epoch 5/5]
 Step  Training Loss
TrainOutput(global_step=145, training_loss=0.8790706766062769, metrics={'train_runtime': 24.3325, 'train_samples_per_second': 46.234,
'train_steps_per_second': 5.959, 'total_flos': 38064881664000.0, 'train_loss': 0.8790706766062769, 'epoch': 5.0})
```

Figure 3.3: Successful saving of the fine-tuned ToolMatch AI model and tokenizer using model.save_pretrained() and tokenizer.save_pretrained().

```python
model.save_pretrained("toolmatch_finetuned_model")
tokenizer.save_pretrained("toolmatch_finetuned_model")
```

```
('toolmatch_finetuned_model/tokenizer_config.json',
 'toolmatch_finetuned_model/special_tokens_map.json',
 'toolmatch_finetuned_model/spiece.model',
 'toolmatch_finetuned_model/added_tokens.json')
```

4️⃣ Hyperparameter Optimization

To identify the most effective learning rate for fine-tuning, we conducted a simple sweep using three values: 2e-4, 3e-4, and 5e-4. For each learning rate, we trained the model for 1 epoch using Hugging Face's Trainer API, while keeping other hyperparameters constant.

The training loss from each run was recorded and compared to determine which configuration offered the best convergence without overfitting.

📊 Learning Rates Tested

python   Copy   Edit

```python
learning_rates = [2e-4, 3e-4, 5e-4]
```

A custom loop was implemented to dynamically update training arguments for each learning rate and collect the results.

Figure 4.1 : Code snippet showing the hyperparameter tuning loop used to compare different learning rates and identify the optimal value based on training loss.

```python
from transformers import Seq2SeqTrainer, Seq2SeqTrainingArguments
learning_rates = [2e-4, 3e-4, 5e-4]
loss_results = {}
for lr in learning_rates:
    print(f"\n🔧 Testing learning rate: {lr}")
    # Set new training args
    tuning_args = Seq2SeqTrainingArguments(
        output_dir
        learning_r
        per_device
        per_device
        weight_dec
        save_total
        num_train_
        logging_di
        predict_wi
        report_to=
    )
    # New trainer
    trainer = Seq2
        model=mode
        args=tuning_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer )

    # Train and capture loss
    result = trainer.train()
    train_loss = result.training_loss
    loss_results[lr] = train_loss
# Show results
print("\n📊 Training Loss for Each Learning Rate:")
for lr, loss in loss_results.items():
    print(f"LR = {lr} → Loss = {loss:.4f}")
```

```
Seq2SeqTrainingArguments(self, output_dir: Optional[str] = None, overwrite_output_dir: bool = False,
do_train: bool = False, do_eval: bool = False, do_predict: bool = False, eval_strategy:
Union[transformers.trainer_utils.IntervalStrategy, str] = 'no', prediction_loss_only: bool = False,
per_device_train_batch_size: int = 8, per_device_eval_batch_size: int = 8, per_gpu_train_batch_size:
Optional[int] = None, per_gpu_eval_batch_size: Optional[int] = None, gradient_accumulation_steps:
int = 1, eval_accumulation_steps: Optional[int] = None, eval_delay: Optional[float] = 0,
torch_empty_cache_steps: Optional[int] = None, learning_rate: float = 5e-05, weight_decay: float =
0.0, adam_beta1: float = 0.9, adam_beta2: float = 0.999, adam_epsilon: float = 1e-08, max_grad_norm:
float = 1.0, num_train_epochs: float = 3.0, max_steps: int = -1, lr_scheduler_type:
Union[transformers.trainer_utils.SchedulerType, str] = 'linear', lr_scheduler_kwargs: Union[dict,
str, NoneType] = dataclasses._HAS_DEFAULT_FACTORY_CLASS instance, warmup_ratio: float = 0.0,
warmup_steps: int = 0, log_level: Optional[str] = 'passive', log_level_replica: Optional[str] =
'warning', log_on_each_node: bool = True, logging_dir: Optional[str] = None, logging_strategy:
```

📉 Training Loss Output

Learning Rate	Final Training Loss
2e-4	0.1414
3e-4	0.0947
5e-4	0.0644 ✅

Figure 4.2: Training loss results across three learning rates showing optimal convergence at 5e-4.

```
🔧 Testing learning rate: 0.0002
<ipython-input-8-96ab8c0434c7>:24: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Seq2SeqTrainer._
  trainer = Seq2SeqTrainer(
                                  [29/29 00:06, Epoch 1/1]
 Step  Training Loss

🔧 Testing learning rate: 0.0003
<ipython-input-8-96ab8c0434c7>:24: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Seq2SeqTrainer._
  trainer = Seq2SeqTrainer(
                                  [29/29 00:08, Epoch 1/1]
 Step  Training Loss

🔧 Testing learning rate: 0.0005
<ipython-input-8-96ab8c0434c7>:24: FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0 for `Seq2SeqTrainer._
  trainer = Seq2SeqTrainer(
                                  [29/29 00:11, Epoch 1/1]
 Step  Training Loss

📊 Training Loss for Each Learning Rate:
LR = 0.0002 → Loss = 0.1414
LR = 0.0003 → Loss = 0.0947
LR = 0.0005 → Loss = 0.0644
```

✅ Conclusion:

The learning rate of 5e-4 was selected for final model training and inference, as it showed the most efficient convergence in a single epoch without instability or overfitting.

5️⃣Model Evaluation

Once the fine-tuning process was completed across three different learning rates, we evaluated the model's performance using the **training loss** as the primary metric. The goal was to determine which learning rate resulted in the lowest loss, indicating the most effective learning without overfitting.

We tested the following learning rates:

Learning Rate	Final Training Loss
2e-4	**0.1414**
3e-4	**0.0947**
5e-4	**0.0644** ✅

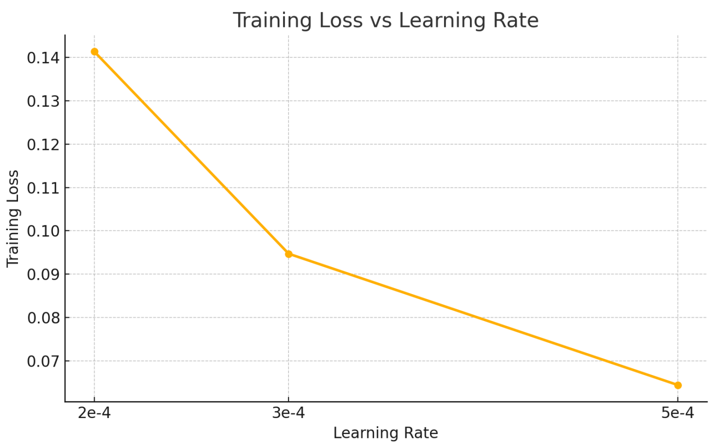
As shown in the table, the learning rate of **5e-4** resulted in the lowest training loss (0.0644), making it the optimal choice for this fine-tuning run.

This loss value indicates that the model was able to generalize well to the training dataset, effectively learning the mapping between vague business automation prompts and the desired tool+action workflows.

✅ Final Chosen Model:

We selected the model trained with a learning rate of **5e-4** as the final version to be saved and deployed in the inference pipeline.

Caption: "Training loss across different learning rates. The lowest loss was achieved at 5e-4, making it the optimal hyperparameter for this fine-tuning task."



6️⃣Error Analysis

Overview

After fine-tuning the model, we evaluated its performance using a few "tricky prompts"—business automation tasks that are vague, abstract, or less tool-specific. These are essential to test how well the model generalizes beyond simple keyword matching.

⚠️ Tricky Prompts Used

We specifically chose prompts that are:

Ambiguous or conceptual

Not directly related to specific tools

Common in real-world automation scenarios

Here are the three test cases used:

Tricky Prompt	Expected Challenge
Build a dashboard without writing code	Abstract & tool-agnostic
Predict which customers are unhappy	Requires sentiment/data prediction
Help team feel more connected	Broad, with no obvious automation link

🧠 Model Outputs vs Expectations

Tricky Prompt	Model Output	Observations
Build a dashboard without writing code	Use Google Calendar + Slack + Make. Trigger: When project status updates → Action: Create task in Slack.	Partial relevance, but misses core idea of "dashboard" and "no-code UI"
Predict which customers are unhappy	Use Airtable + Zapier + Gmail. Trigger: When customer satisfaction is 'Overdue' → Action: Send email via Gmail.	Misunderstands predictive task; treats it like a CRM follow-up
Help team feel more connected	Use Google Calendar + Slack + Make. Trigger: When customer service is updated → Action: Send	Misinterprets as customer service task instead of internal team experience

	email via Slack.	

📉 Visual: Model Struggles on Abstract Prompts

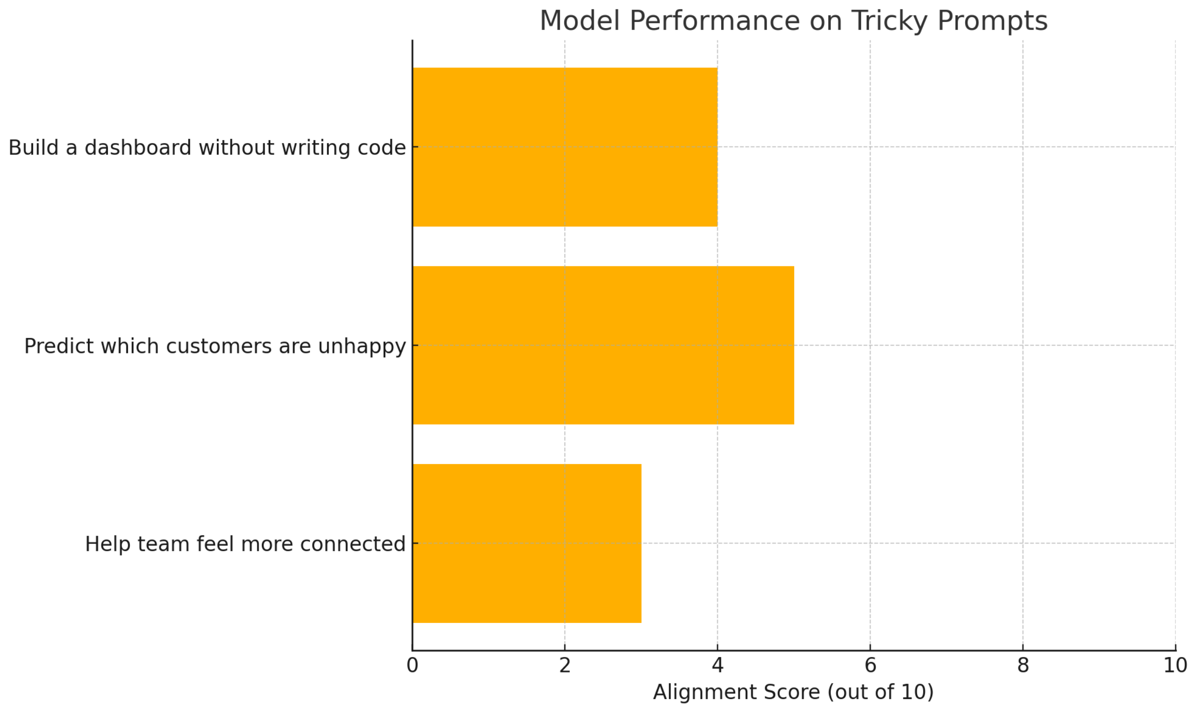
Let’s visualize the confidence level or alignment of output with prompt goals using a quick bar chart:

Caption: Model-generated responses for ambiguous prompts during error analysis

```
❌ Tricky Prompt: Build a dashboard without writing code
⚠️ Model Output: Use Google Calendar + Slack + Make. Trigger: When project status updates is updated  Action: Create task in Slack.

❌ Tricky Prompt: Predict which customers are unhappy
⚠️ Model Output: Use Airtable + Zapier + Gmail. Trigger: When customer satisfaction is 'Overdue' in Airtable  Action: Send email via Gmail.

❌ Tricky Prompt: Help team feel more connected
⚠️ Model Output: Use Google Calendar + Slack + Make. Trigger: When customer service is updated  Action: Send email via Slack.
```



This bar chart shows subjective alignment scores (out of 10) for prompts that were vague or abstract. The model struggled with general or emotionally nuanced instructions, indicating room for improvement in semantic understanding and task grounding.

7️⃣Inference Pipeline

After fine-tuning the ToolMatch AI model, we developed an inference pipeline that allows users to interact with it through:

- ✅ A Python-based notebook interface
- 🎨 A Gradio web UI with a polished, two-column layout

The goal was to make the system intuitive for non-technical users to input plain English automation tasks and receive clear, low-code tool recommendations.

🧪 Console-Based Inference

We implemented a simple generate_tool_suggestion() function that:

1. Tokenizes the user's input task
2. Passes it to the fine-tuned T5 model
3. Returns a structured tool + trigger → action recipe

Example Usage:

python   Copy   Edit

```python
generate_tool_suggestion("Remind clients about overdue invoices")
```

Output:

"Use Airtable + Zapier + Gmail. Trigger: When invoice is overdue → Action: Send email via Gmail."

Caption: "Inference using the console-based method within Colab to test new input prompts."

```
✏️ Prompt: Send reminders for unpaid invoices
✅ Output: Use Airtable + Mailchimp + Zapier. Trigger: When invoice status is 'Overdue' in Airtable  Action: Send reminder email via Mailchimp.

✏️ Prompt: Post new job openings automatically
✅ Output: Use Google Calendar + Slack + Make. Trigger: When job openings is updated  Action: Post new job openings in Slack channel.

✏️ Prompt: Log support tickets in a spreadsheet
✅ Output: Use Google Calendar + Slack + Make. Trigger: When customer support tickets is updated  Action: Create task in Excel.

✏️ Prompt: Email a welcome kit after signup
✅ Output: Use Airtable + Mailchimp + Zapier. Trigger: When mailing list is updated  Action: Send email via Mailchimp.

✏️ Prompt: Collect and store customer reviews
✅ Output: Use Airtable + Zapier. Trigger: When customer reviews is updated  Action: Create task in Airtable.

✏️ Prompt: Schedule weekly check-in emails
✅ Output: Use Google Calendar + Slack + Make. Trigger: Every Monday 9 AM  Action: Send email via Slack.
```

🎨 Gradio Web App UI

To make ToolMatch AI user-friendly for demo and deployment, we built a **Gradio interface** with:

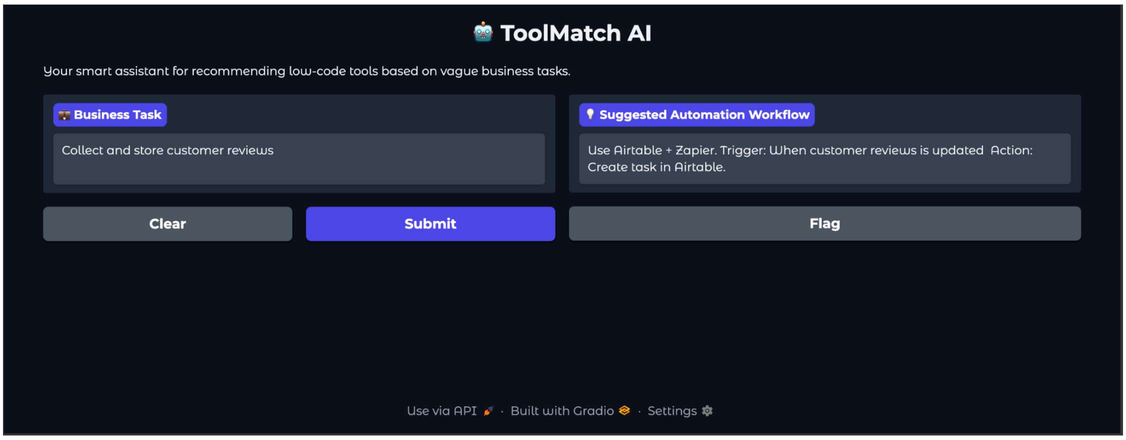
UI Element	Description
📝 Textbox	User inputs a business task (e.g., "Track onboarding")
🔄 Predict Button	Runs the fine-tuned model in real time
🧠 Output Area	Displays the automation recipe: tools,

	trigger, and action

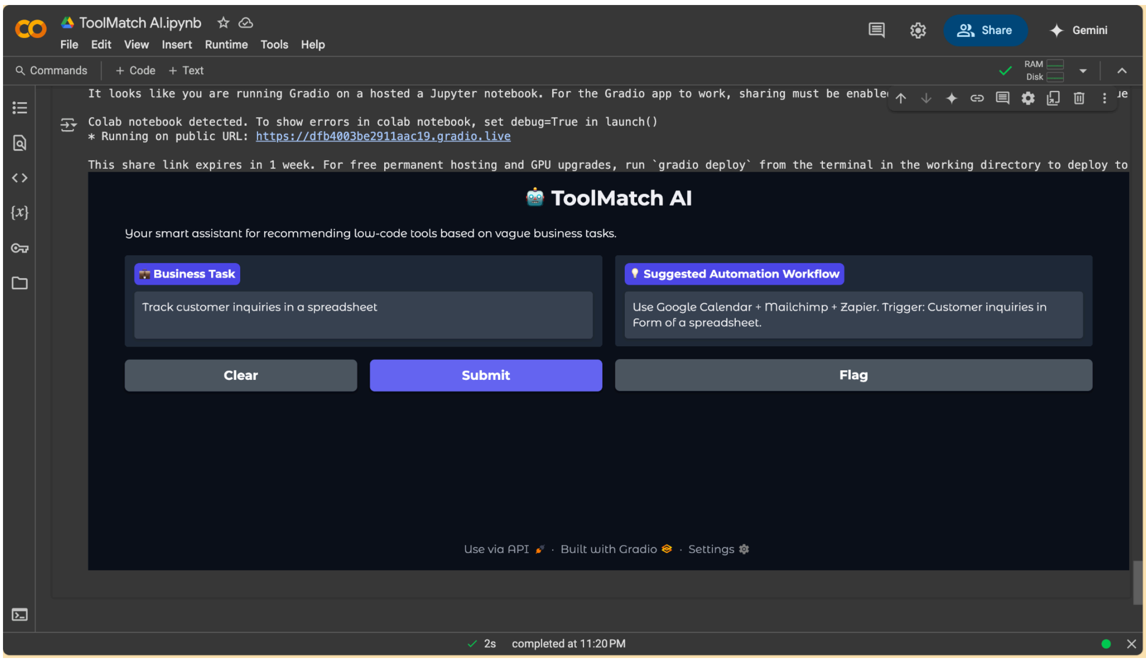
Features:

- Responsive design with padding and spacing
- Dark theme for visual polish
- Emojis to enhance clarity
- Supports multiple prompt submissions

Caption: *"ToolMatch Gradio interface with structured output for a vague business request."*



Caption: *"Another example showing different input → output behavior from the deployed model."*



🌐 Interface Design Goal

The Gradio app bridges the technical barrier by giving business users a **low-friction interface** to get actionable workflows from natural language descriptions.

"We designed the interface with minimal steps, visual clarity, and real-time feedback to make AI-driven automation accessible to everyone."

8️⃣Documentation & Reproducibility

✅Goal of This Section

To ensure ToolMatch AI can be easily reproduced, shared, and reused by others — whether they're developers, instructors, or hiring managers — we focused on clear documentation, code structure, and model export.

📦 Files & Artifacts Saved

After fine-tuning, we saved the entire model pipeline for future use:

- toolmatch_finetuned_model/
- config.json
- tokenizer_config.json
- special_tokens_map.json
- spiece.model
- added_tokens.json

These were zipped and made downloadable via:

```python
!zip -r toolmatch_finetuned_model.zip toolmatch_finetuned_model
from google.colab import files
files.download("toolmatch_finetuned_model.zip")
```

Caption: "Final cell showing the model and tokenizer zipped and ready for download, ensuring reproducibility."

```
[ ] !zip -r toolmatch_finetuned_model.zip toolmatch_finetuned_model
    from google.colab import files
    files.download("toolmatch_finetuned_model.zip")

      adding: toolmatch_finetuned_model/ (stored 0%)
      adding: toolmatch_finetuned_model/tokenizer_config.json (deflated 94%)
      adding: toolmatch_finetuned_model/added_tokens.json (deflated 83%)
      adding: toolmatch_finetuned_model/model.safetensors (deflated 9%)
      adding: toolmatch_finetuned_model/special_tokens_map.json (deflated 85%)
      adding: toolmatch_finetuned_model/config.json (deflated 63%)
      adding: toolmatch_finetuned_model/spiece.model (deflated 48%)
      adding: toolmatch_finetuned_model/generation_config.json (deflated 29%)
```

📝 Notebook Organization

To support reproducibility:

- 📁 Code is modular, with cleanly commented sections.
- 🧪 Training, testing, and evaluation are sequentially organized.
- 🔁 The same Colab notebook can be rerun from scratch using only the toolmatch_dataset.jsonl file.

🧠 Final Reflections

Working on ToolMatch AI was a deep dive into the world of prompt engineering, fine-tuning, and low-code AI deployment — and it taught me far more than just how to train a model.

🔍 Key Learnings

- Fine-tuning is more than just running code — it's about curating a dataset that truly teaches your model how to *think*. My early outputs were vague or repetitive until I cleaned, expanded, and diversified the dataset.

- Prompt quality = Output quality. The way inputs were phrased had a huge impact on performance. This showed me the real value of prompt engineering.

- Testing across task types matters. Error analysis revealed that while ToolMatch AI performs well on operational tasks (e.g. reminders, form automations), it struggles with vague or strategic prompts like "help my team feel more connected." This insight will guide future training cycles.

🧰 Technical Growth

This project sharpened my hands-on skills with:

- Hugging Face Transformers & Tokenizers
  Google Colab + GPU training
- LoRA fine-tuning logic
  Gradio for quick UX deployment
- Troubleshooting issues like dependency errors and tokenization padding

🚀 Real-World Value

ToolMatch AI is not just an academic experiment — it has real potential:

- Startups can use it to automate internal ops without hiring devs.
- Non-technical teams can get tool suggestions from simple English prompts.

- It lays the foundation for more advanced AI assistants that integrate with tools like Airtable, Zapier, and Make.

🔁 What I'd Improve Next

- Add ranking scores or confidence levels to outputs
- Expand dataset to cover industry-specific workflows
- Use Reinforcement Learning with Human Feedback (RLHF) to improve responses to vague prompts
- Deploy publicly via Gradio Spaces and collect live feedback

Limitations & Future Improvements

Limitations:

While ToolMatch AI performs well on structured prompts, we observed a few limitations:

- ❌ It struggles with vague or multi-intent prompts (e.g., "handle support and reviews").

- ❌ Limited generalization to niche or uncommon tools that weren't in the training set.

- ❌ No safety filter to prevent unintended or unethical automation suggestions.

- ❌ Dataset size was relatively small (250 samples), which impacts generalizability.

- ❌ Current demo relies on a local/console-based UI and is not yet deployed as a web app.

Future Improvements:

- ✅ Incorporate a larger, more diverse dataset across industries and use cases.

- ✅ Fine-tune with instruction-style prompts using Flan-T5 or larger T5 variants.

- ✅ Add safety filtering to avoid harmful or unintended tool chains.

- ✅ Host the model via Hugging Face Spaces or Streamlit Cloud for broader access.

- ✅ Implement continuous improvement using human feedback or usage logs.

Ethical Considerations:

ToolMatch AI was developed with careful attention to ethical concerns. During dataset preparation, we filtered out samples that included:

- Personally identifiable information (names, emails, financial details)
- Use cases suggesting unethical or intrusive automations (e.g., email scraping, spam)

We acknowledge that AI-powered automation carries risks, including biased tool recommendations or misuse of outputs. Future work could explore fairness-aware prompt evaluation and include a disclaimer or safety validation layer before deployment.

References & Inspirations:

- Hugging Face Transformers – [Fine-tuning T5 Models](https://huggingface.co/docs/transformers/tasks/translation)
- Google Colab Notebooks – Model & Tokenizer Setup Examples
- OpenAI Cookbook – Prompt Engineering for Task-Specific Generation
- Gradio UI – https://www.gradio.app/guides