CHAINS in depth.

why use chains:

An LLM-based application is built from **multiple smaller steps**, like formatting a prompt, calling the model, and processing the response.

Chains help you organize these steps into a smooth, connected **pipeline** where each step flows into the next.

Chains make this process easier by **automatically passing the output of one step as the input to the next**, allowing us to build complex workflows in a clean and structured way.

Types of chains:

1. Simple chains
2. Parallel chain
3. Sequential chain
4. Conditional chain

simple 3 step chain:

1. input from user → prompt
2. send prompt to llm
3. display llm response in proper format.

code :

```python
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser

load_dotenv()
#1.create a dynamic prompt template.
prompt = PromptTemplate(
    template='Generate 5 interesting facts about {topic}',
    input_variables=['topic']
)

llm = HuggingFaceEndpoint(
   repo_id="Qwen/Qwen3-Coder-480B-A35B-Instruct",
   task="text-generation"
)
#2.Create chat model
model = ChatHuggingFace(llm=llm)

#3.output parse to give us output in string format.
parser = StrOutputParser()

#create chain.
chain = prompt | model | parser

#invoke response for query.
result = chain.invoke({'topic':'cricket'})

print(result)

#to visulaize our chain.
#pip install grandalf
chain.get_graph().print_ascii()
```

```
Here are 5 fascinating facts about cricket:

1. **Cricket was once considered a form of gambling** - In the 18th century, betting
on cricket matches was so prevalent that the sport was often associated with gambling
 dens and wagering culture, leading to various attempts to regulate and clean up the
game.

2. **The longest cricket match lasted 12 days** - The timeless Test match between Eng
land and South Africa in 1939 at Durban is famous for ending in a draw after 12 days
of play, with England needing just 7 runs to win but time running out due to a packed
 shipping schedule.

3. **Cricket is played in zero gravity** - In 1992, British astronaut Michael Foale b
ecame the first person to play cricket in space aboard the Space Shuttle Discovery, h
itting a ball (attached to a string) while floating in microgravity.

4. **The word "wicket" comes from "wicki-up"** - The term originates from old English
 meaning "dwelling" or "tent," referring to the small huts or shelters that early cri
cketers would build behind the stumps for protection.

5. **Australia once fielded an entire team of convicts** - In 1829, an Australian tea
m playing against a visiting English side was composed entirely of emancipated convic
ts, yet they still managed to compete credibly against their more formally trained op
ponents.
```

To visualize our chain use: chain.get_graph().print_ascii()

```
  +------------+
  | PromptInput |
  +------------+
        *
        *
        *
 +---------------+
 | PromptTemplate |
 +---------------+
        *
        *
        *
+----------------+
| ChatHuggingFace |
+----------------+
  +------------+
  | PromptInput |
  +------------+
        *
        *
        *
 +---------------+
 | PromptTemplate |
 +---------------+
        *
        *
        *
+----------------+
| ChatHuggingFace |
+----------------+
        *
+----------------+
| ChatHuggingFace |
+----------------+
        *
        *
```

2.Sequential Chain:

A **SequentialChain** in LangChain runs a **series of steps one after the other**, where **each step's output becomes the next step's input**. Think of it like a relay race each runner (step) passes the baton (output) to the next, creating a smooth, automatic flow of logic.

example:

You want your AI to help a student study a topic in this flow:

1. **Take a topic** (e.g., *Photosynthesis*)
2. **Generate a summary** of the topic
3. **Generate 3 quiz questions** from that summary

Why This is Useful

- **Multi-step tasks** like research → summary → quiz/test prep
- Reuse **intermediate outputs** (summary) in later tasks
- Everything stays structured and connected

code example:

4.
```
prompt1 = PromptTemplate(
    template='Generate a summary on {topic}',
    input_variables=['topic']
)

prompt2 = PromptTemplate(
    template='Generate a 3 quiz questions from the following text \n {text}',
    input_variables=['text']
)

llm = HuggingFaceEndpoint(
   repo_id="Qwen/Qwen3-Coder-480B-A35B-Instruct",
   task="text-generation"
)

# Create chat model
model = ChatHuggingFace(llm=llm)

parser = StrOutputParser()

chain = prompt1 | model | parser | prompt2 | model | parser

result = chain.invoke({'topic': 'Phototsynthesis'})

print(result)
```

prompt1 :takes the initial input and turns it into a structured prompt. model uses that prompt to generate a response. parser: cleans or extracts useful information from the model's output. prompt2: takes that parsed info and creates a second prompt. model then generates a second response using that new prompt. parser: processes the final output to extract the final result.

OUTPUT:

```
Here are 3 quiz questions based on the photosynthesis text:

**Question 1:** What are the two main stages of photosynthesis, and where does each s
tage occur within the chloroplast?

**Question 2:** In the basic photosynthesis equation, what are the three reactants ne
eded to produce glucose and oxygen?

**Question 3:** Why is photosynthesis considered fundamental to life on Earth? List a
t least three important reasons mentioned in the text.

---

**Answer Key:**

**Question 1:**
- Light-dependent reactions (occur in thylakoid membranes)
- Light-independent reactions/Calvin Cycle (occur in the stroma)

**Question 2:**
Carbon dioxide (CO2), water (H2O), and light energy

**Question 3:**
Produces oxygen essential for most life, forms the base of most food chains, removes
CO2 from atmosphere, and converts solar energy into usable chemical energy
(venv) PS C:\Users\shree\Desktop\Shreesha\Langchain_code\5.chains>
```

3.Parallel chains:

Parallel Chains in LangChain are used when you want to run multiple chains at the same time (in parallel) using the same or different inputs. Each chain does its own task separately, and their outputs are combined at the end.

Think of it like asking 3 friends different questions at the same time you wait and gather all their answers together.

Example:

You're studying for an exam and ask two friends for help at the same time:

- **Friend 1** reads your textbook chapter and writes short, simple notes.
- **Friend 2** reads the same chapter and creates 5 quick quiz questions.

Both friends do their tasks independently and hand you their results.

You then take both the notes and quiz and ask **Friend 1** to merge them into one clean study sheet.

This is exactly what a **parallel chain followed by a merge chain** does in LangChain:
It runs multiple tasks at once (in parallel), then combines the results into one (merge).

FRIEND== MODEL.

code :

```python
#Step 1: Define the LLMs (models)
llm1 = HuggingFaceEndpoint(
    repo_id="Qwen/Qwen3-Coder-480B-A35B-Instruct",  #or use ChatOpenAI()
    task="text-generation"
)

model1 = ChatHuggingFace(llm=llm1)

llm2 = HuggingFaceEndpoint(
    repo_id="mistralai/Mistral-7B-Instruct-v0.3",  #or use claude.
    task="text-generation"
)

model2 = ChatHuggingFace(llm=llm2)
# Step 2: Define PromptTemplates
# Prompt to generate notes from the input text
prompt1 = PromptTemplate(
     template='Generate short and simple notes from the following text \n {text}',
     input_variables=['text']
)
# Prompt to generate questions (quiz) from the input text
prompt2 = PromptTemplate(
     template='Generate 5 short question answers from the following text \n {text}',
     input_variables=['text']
)
# Prompt to merge both notes and quiz into a final document
prompt3 = PromptTemplate(
     template='Merge the provided notes and quiz into a single document \n notes -> {notes} and quiz -> {quiz}',
     input_variables=['notes', 'quiz']
)
```

```python
parser = StrOutputParser()

# Step 3: Create Chains
# Run notes and quiz generation in parallel
parallel_chain = RunnableParallel({
    'notes': prompt1 | model1 | parser,
    'quiz': prompt2 | model2 | parser
})

# Merge the results of notes and quiz into a final output
merge_chain = prompt3 | model1 | parser
# Full pipeline: parallel → merge
chain = parallel_chain | merge_chain
# Step 4: Input Text
text = """
Support vector machines (SVMs) are a set of supervised learning
The advantages of support vector machines are:
Effective in high dimensional spaces.
Still effective in cases where number of dimensions is greater
Uses a subset of training points in the decision function (cal
Versatile: different Kernel functions can be specified for the
The disadvantages of support vector machines include:
If the number of features is much greater than the number of sa
SVMs do not directly provide probability estimates, these are
The support vector machines in scikit-learn support both dense
"""
#step 5: Invoke the chain with input
result = chain.invoke({'text':text})

print(result)
```

output:

```
(venv) PS C:\Users\shree\Desktop\Shreesha\Langchain_code\5.chains> python 3.parallel.py
# Support Vector Machines (SVM) - Complete Guide

## **What is SVM?**
- Supervised learning method
- Used for: Classification, Regression, Outlier detection
- A set of supervised learning methods used for classification, regression, and outliers detection

## **Advantages**
- ✅ Works well in high-dimensional spaces
- ✅ Effective when dimensions > samples
- ✅ Memory efficient (uses only support vectors)
- ✅ Versatile (different kernel functions available)
- ✅ Effective in high dimensional spaces, still effective in cases where the number of dimensions is greater than the number of samples
- ✅ Uses a subset of training points in the decision function (called support vectors)
- ✅ Is versatile

## **Disadvantages**
- ❌ Risk of over-fitting when features >> samples
- ❌ No direct probability estimates (requires expensive cross-validation)
- ❌ If the number of features is much greater than the number of samples, over-fitting in choosing Kernel functions and the regularization term is crucial
```

```
## **Technical Details**
- **Input support**: Dense (numpy.ndarray) and sparse (scipy.sparse) data
- **Best performance**: C-ordered numpy.ndarray or scipy.sparse.csr_matrix
- **Data type**: float64 recommended
- **scikit-learn support**: scikit-learn supports both dense (numpy.ndarray and convertible to that by numpy.asarray) and sparse (any scipy.sparse) sample vectors as input, but when m
aking predictions for sparse data, the SVM must have been fit on such data

## **Probability Estimates**
- SVMs do not directly provide probability estimates
- These are calculated using an expensive five-fold cross-validation

---

## **Quiz Questions**

1. **What is a support vector machine (SVM)?**
   Answer: A set of supervised learning methods used for classification, regression, and outliers detection.

2. **What are the advantages of using a support vector machine (SVM)?**
   Answer: Effective in high dimensional spaces, still effective in cases where the number of dimensions is greater than the number of samples, uses a subset of training points in the
 decision function (called support vectors), and is versatile.

3. **What is a disadvantage of using a support vector machine (SVM)?**
   Answer: If the number of features is much greater than the number of samples, over-fitting in choosing Kernel functions and the regularization term is crucial.

4. **How does a support vector machine (SVM) provide probability estimates?**
   Answer: SVMs do not directly provide probability estimates. These are calculated using an expensive five-fold cross-validation.

5. **What types of sample vectors does scikit-learn support for support vector machines (SVMs)?**
   Answer: scikit-learn supports both dense (numpy.ndarray and convertible to that by numpy.asarray) and sparse (any scipy.sparse) sample vectors as input, but when making predictions
 for sparse data, the SVM must have been fit on such data.
```

chain visualization :

```
            +--------------------------+
            | Parallel<notes,quiz>Input |
            +--------------------------+
               ***             ***
             ***                 ***
           **                       **
+----------------+           +----------------+
| PromptTemplate |           | PromptTemplate |
+----------------+           +----------------+
         *                            *
         *                            *
         *                            *
+-----------------+          +-----------------+
| ChatHuggingFace |          | ChatHuggingFace |
+-----------------+          +-----------------+
         *                            *
         *                            *
         *                            *
+-----------------+          +-----------------+
| StrOutputParser |          | StrOutputParser |
+-----------------+          +-----------------+
               ***             ***
                  ***       ***
                     **   **
           +---------------------------+
           | Parallel<notes,quiz>Output |
           +---------------------------+
                         *
                         *
                         *
                 +----------------+
                 | PromptTemplate |
                 +----------------+
                         *
```

```
                *
        +----------------+
        | PromptTemplate |
        +----------------+
                *
                *
                *
       +-----------------+
       | ChatHuggingFace |
       +-----------------+
                *
                *
                *
       +-----------------+
       | StrOutputParser |
       +-----------------+
                *
                *
                *
    +-----------------------+
    | StrOutputParserOutput |
    +-----------------------+
```

RunnableParallel takes a dictionary of keys → chains. It runs all the chains **in parallel** and returns a result where the keys match your original dictionary.

So it’s like:

pythonCopy codenotes_output = notes_chain.invoke(input) quiz_output = quiz_chain.invoke(input) return {'notes': notes_output, 'quiz': quiz_output}

but done **more efficiently**, and automatically.

Conditional chains:

Conditional chains in LangChain let you choose what to do next based on a condition. Instead of always running steps in order, you check a value and pick a path. This is helpful when your response depends on input—like positive or negative feedback. RunnableBranch is used to handle these choices, just like an if-else ladder. It makes your logic more flexible and easier to manage.

Example Scenario: Handling Positive and Negative Feedback

Let’s say you are building a customer feedback responder. You want the system to:

- **Detect if feedback is positive or negative**
- **Send a tailored response based on that sentiment**

To implement this in LangChain:

1. You first **classify the sentiment** using a model and a PydanticOutputParser
2. Based on whether the sentiment is **positive or negative**, the response chain changes dynamically
3. A RunnableBranch routes the input to the appropriate response prompt

code:

```python
model = ChatHuggingFace(llm=llm)

# Output parser to return plain text
parser = StrOutputParser()

# Define structured output schema for sentiment classification
class Feedback(BaseModel):
    sentiment: Literal['positive', 'negative'] = Field(description='Give the sentiment of the feedback')

# Structured parser using the Feedback schema
parser2 = PydanticOutputParser(pydantic_object=Feedback)

# Prompt to classify feedback sentiment
prompt1 = PromptTemplate(
    template='Classify the sentiment of the following feedback text into postive or negative \n {feedback} \n {format_instruction}',
    input_variables=['feedback'],
    partial_variables={'format_instruction': parser2.get_format_instructions()}
)

# Create a classification chain: prompt → model → structured parser
classifier_chain = prompt1 | model | parser2

# Prompt to generate response to positive feedback
prompt2 = PromptTemplate(
    template='Write an appropriate response to this positive feedback \n {feedback}',
    input_variables=['feedback']
)
```

```python
# Create a classification chain: prompt → model → structured parser
classifier_chain = prompt1 | model | parser2

# Prompt to generate response to positive feedback
prompt2 = PromptTemplate(
    template='Write an appropriate response to this positive feedback \n {feedback}',
    input_variables=['feedback']
)

# Prompt to generate response to negative feedback
prompt3 = PromptTemplate(
    template='Write an appropriate response to this negative feedback \n {feedback}',
    input_variables=['feedback']
)

# Branch chain to choose response path based on sentiment
branch_chain = RunnableBranch(
    (lambda x: x.sentiment == 'positive', prompt2 | model | parser),   # If positive, use positive prompt
    (lambda x: x.sentiment == 'negative', prompt3 | model | parser),   # If negative, use negative prompt
    RunnableLambda(lambda x: "could not find sentiment")               # Fallback if no match
)

# Final pipeline: classify first, then respond based on classification
chain = classifier_chain | branch_chain

# Test the chain with a sample feedback
print(chain.invoke({'feedback': 'This is a beautiful phone'}))
```

output:

```
(venv) PS C:\Users\shree\Desktop\Shreesha\Langchain_code\5.chains> python 4.conditional.py
Thanks a bunch for the kind words! I'm here to help you with your questions, so don't hesitate to ask.
```